Tool Support for the Goal-Oriented Requirement Language

Yi Chu

Project Report submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements for the degree of

Master of Computer Science

Under the auspices of the Ottawa-Carleton Institute for Computer Science

University of Ottawa
Ottawa, Ontario, Canada
August 2005

Abstract

The Goal-Oriented Requirement Language (GRL) is a new and evolving notation used to specify and analyze goals and requirements. In this project, we develop a software tool (also called *GRL editor*) that supports this language. The Generic Modeling Environment (GME) is a framework used to create domain-specific modeling environments, and thus can be the platform for developing our GRL Editor. This report gives a thorough discussion of GME functionalities and the feasibility of using it for creating a GRL editor.

The GRL meta-model is defined as a UML class diagram, and is implemented by a GME meta-model. The translation output of this meta-model is the GRL editor. The details of implementing this meta-model, along with the decision-making process are also part of this report.

Due to the need for a better visual representation of GRL with GME, a limited amount of COM programming was performed in this project. This report gives a brief introduction to this work, with a focus on laying out a framework for future development.

Besides developing the tool based on the current GRL definition, this report describes some considerations on other important factors such as the extensibility and maintainability of the tool when the meta-model for GRL evolves in the future. Comparisons are made to other tools that support or could support GRL model editing. Model evaluation and interpretation are also discussed briefly.

Acknowledgment

Many thanks to Dr. Daniel Amyot who supervised this project work and provided numerous suggestions and advices that were key to the successful completion of this project. It is really a valuable experience to carry out this research with his help. I am so happy that I have been trained with his rigorous approach – the experience is rewarding.

I would like to thank my wife, Yun Xiao, for the support of my research work she has given during the four years of my study. Your help and support is part of the foundation based on which this project is carried out.

Table of Contents

List of Figures

Glossary

Term	Definition
BNF	Backus Normal Form
COM	Component Object Model
EMF	Eclipse Modeling Framework
FCO	GME concept for First Class Object
GEF	Graphical Editing Framework
GME	Generic Modeling Environment developed by Vanderbilt University
GRL	Goal-oriented Requirements Language
GRL Editor	A GME paradigm that supports creating and editing GRL models
Meta-model	A model that defines constructs used to create models
MGA	MultiGraph Architecture, GME Component for creating Modol-integrated Program Synthesis environment
MOF	Meta-Object Facility Specification, an OMG standard
NFR	Non-Functional Requirements
OCL	Object Constraint Language
OME	Organization Modeling Environment
Paradigm	A GME environment that supports modeling concepts, the rules for constructing the model, and the visual editing of these concepts
QVT	Query View Transformation, an OMG standard
UCM	Use Case Maps
UML	Unified Modeling Language
URN	User Requirements Notation
XMF	XMF-Mosaic, a platform for building modeling tools
XML	Extensible Markup Language

Chapter 1. Introduction

The *Goal-oriented Language* (GRL) is a modeling language currently being standardized by the International Telecommunications Union as part of the User Requirements Notation (URN) [1] [16]. For the last decade, goal-oriented modeling has been a very active field in the requirements engineering community [24]. One well-established language is the *NFR framework*, published in [3]. GRL includes some of the most interesting concepts found in the NFR framework and complements them with agent modeling concepts from the *i** framework [22]. GRL captures business or system goals, alternative means of achieving goals, and the rationale for goals and alternatives. The notation, summarized in Figure 1 is applicable to non-functional as well as functional requirements. It has been used in various domains including information systems [14], telecommunication systems [1], and business processes modeling [23].

In this project, we develop a graphical editor (also called a GRL Editor) supporting the GRL notation and enabling the analysis of GRL models. The editor will be based on a GRL *meta-model*. A meta-model captures the concepts and relationships of a notation or language. It is often expressed as a UML class diagram [15]. The GRL meta-model used in this project is a simplified meta-model for a subset of the GRL notation, as shown in Figure 5.

1.1. Approach

A GRL Editor supports a group of notation elements, the rules for setting up a model using these elements, and the evaluation of the model. It also supports modeling of very complex system and thus reuse of models and extensibility of the models are very important factors in designing such an editor. Instead of building an editor from scratch in some programming language, we will explore the use of the *Generic Modeling Environment* (GME), which provides facilities for the creation of visual editors whose syntax is formalized as a meta-model [5]. Such environment promises to accelerate the development

of our GRL editor, especially as the GRL meta-model is not yet finalized and is likely to evolve in the near future.

The goal of this project is to study the functionality of GME and develop a GRL editor using GME. GME's MultiGraph Architecture (MGA) [5] supports graphical creation of domain specific modeling paradigm based on a meta-model. A GME GRL Editor supports graphical editing of a GRL model, in addition to model analysis and evaluation. Since the GRL meta-model is not finalized, this GRL editor is developed based on a simplified version of GRL meta-model.

Due to the time constraint, model analysis and evaluation using this GRL editor is not included in this project. But, it should be confirmed that this GRL editor is able to be extended to include these functionalities.

1.2. Contributions

The report emphasizes the following contributions:

- Study on the functionalities of GME and the feasibility of using GME to develop a GRL editor.
- Design of GRL editor via a GME-based meta-model.
- GRL editor visualization enhancement via COM programming on the decorators.
- Meta-model evolution experiments that study the effects of meta-model changes on existing models.

1.3. Outline

This report introduces some background knowledge of GRL and GME in the first part. In the second part, it describes the development process of the GRL editor, including some discussions on the feasibility of the designs. This is followed by some discussions on the extensibility and compatibility of the GRL editor. Since the visual presentation of the GME is not sufficient for supporting the GRL notation, an enhancement to the GME Decorator, with COM programming has to be performed. The structure of the report is as follows.

- Chapter 2 – Background knowledge introduction to GRL and GME.
- Chapter 3 – Detailed descriptions about the process of designing a GRL editor.
- Chapter 4 – Visualization enhancement of the GRL editor with COM programming on Decorators.
- Chapter 5 – A complete tutorial of creating a GRL model with the GRL editor.
- Chapter 6 – Introduction to the meta-model evolution experiments that study the effects of meta-model changes on existing models.
- Chapter 7 – Some discussions on the related works and future improvements on this GRL editor, and future work such as model evaluation.
- Chapter 8 – Conclusion.

Chapter 2. Background

This chapter presents some fundamental knowledge and concepts about GRL and GME. These concepts are necessary for understanding the discussions in the following chapter, where the design of GRL editor is described.

2.1. Introduction to GRL

There are three main categories of concepts in GRL [1] [16]: intentional elements, links and actors. The intentional elements are goal, softgoal, task, belief, and resource. The links are contribution, dependency, means end, and decomposition. There are other concepts such as satisfaction levels when a model is evaluated. The GRL concrete textual syntax can be defined in BNF or XML forms. The abstract definition of a GRL meta-model is not yet complete.

2.1.1 GRL Concepts

GRL is made up of the following concepts. For a graphical notation of all the concepts in GRL, see Figure 1.

- **Intentional Elements** - GRL intentional elements are used to specify concepts of why the system is constructed in a certain way, what the alternatives are and how they are chosen.
  - *Goal* - A goal is a condition or state of affairs in the world that the stakeholders would like to achieve.
  - *Softgoal* - A softgoal is a condition or state of affairs in the world that the actor would like to achieve, but unlike in the concept of (hard) goal, there are no clear-cut criteria for whether the condition is achieved, and it is up to subjective judgment and interpretation of the developer to judge whether a particular state of affairs in fact achieves sufficiently the stated softgoal.

- *Task* - A task specifies a particular way of doing something. It is a solution to the system that satisfies certain softgoals.
- *Belief* - Belief represents a design rationale. Beliefs make it possible for domain characteristics to be considered and properly reflected into the decision making process, hence facilitating later review, justification and change of the system, as well as enhancing traceability.
- *Resource* - A resource is a (physical or informational) entity, with which the main concern is whether it is available.

- **Links** - A link (or relationship) connects two elements in a model. It represents a relationship between elements. There are four types of links: Contribution, Decomposition, Means-End and Dependency. *Contribution* links specify how one element contributes to others in a model – the most important type of link. A Contribution link has the following types:
  - *Break* – sufficient to break the goal
  - *Hurt* – some aspects of the goal is hurt but insufficient to break the goal
  - *Some-* – hesitate between Break and Hurt
  - *Unknown* – effect on goal is not yet known
  - *Make* – sufficient to achieve the goal
  - *Help* – helpful to achieve some aspects of the goal but insufficient to achieve the whole goal
  - *Some+* – hesitate between Make and Help
  - *Equal* – equivalent to
- **Actors** - An actor is an active entity that carries out actions to achieve goals by exercising its know-how. Graphically, an actor may optionally have a boundary, with intentional elements inside.
- **Satisfaction Levels** - Satisfaction levels are assigned to intentional elements after the model is evaluated. They represent degrees of satisfaction of intention to be achieved. They have the following types: Satisficed, Weakly Satisficed, Undecided, Weakly Denied, Denied and Conflict.

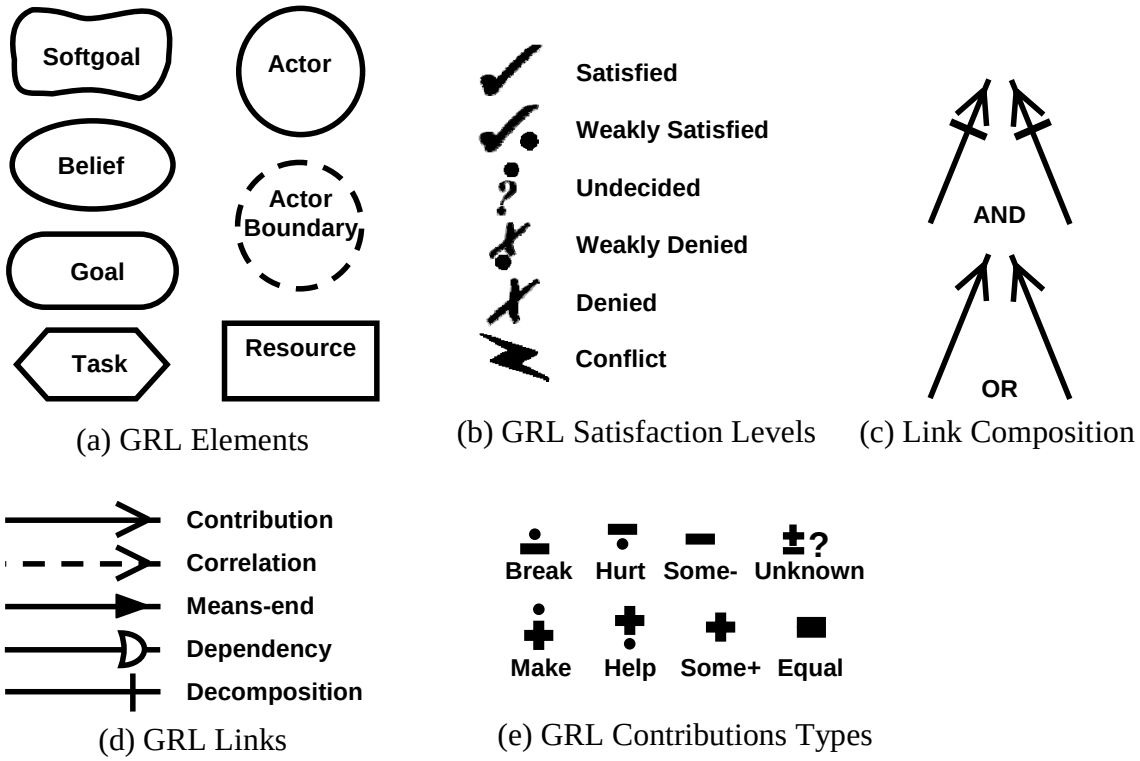


Figure 1 Summary of GRL notations

2.1.2 GRL Syntaxes

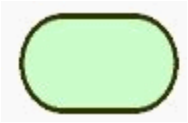
The GRL syntax defines how to present concepts, as described in the previous section. It also defines how to organize the elements to construct the model. The GRL syntax comes in forms [16].

- *Textual form using BNF* - It uses BNF to specify GRL syntax. For example, the following is a BNF definition of Goal:.

```
<Goal> ::= GOAL<Goal Name> [<Informal Textual Description>]
                           [ATTRIBUTE <Attributes>]
                           [OWNER <Actor Name>]
<Attributes > ::= <Attribute > { <Attribute> }
<Attribute> ::= <Element Name>
```

- *Graphical Form* - It uses both BNF and a graphic symbol to represent a concept in GRL. The following is an example for definition of Goal, taken from the KM lab of University of Toronto.

```
<Goal> ::= <Goal Symbol>
                          CONTAINS  <Goal Name > [Attributes]
<Goal Symbol> ::=
```



- *Textual form using XML* – It uses XML elements and attributes to describe the GRL syntax. The following is an example from the XML Document Type Definition.

```
<!ELEMENT model-name EMPTY>
<!ATTLIST model-name
            model-id          ID              #REQUIRED
            name              CDATA           #IMPLIED
            description       CDATA           #IMPLIED>
```

2.1.3 Abstract Definition of GRL Syntaxes - GRL Meta-model

An abstract definition of GRL meta-model can be defined by a UML class diagram as shown in Figure 5. Notice that this is a temporary simplified version of the GRL meta-model definition.

2.1.4 GRL Model Evaluation

GRL model evaluation is used to measure the impact of qualitative decisions on the level of satisfaction of high-level goals. Such mechanism requires one to assign a qualitative degree of satisfaction or availability to tasks and goals at the bottom of the model hierarchy, and then to use a propagation algorithm to compute how well higher-level goals are satisfied. The result of the evaluation is a model within it all the intentional elements are assigned with a certain value of satisfaction level.

2.2. Introduction to GME

The Generic Modeling Environment 4.0 (GME) is developed by the Institute for Software Integrated System, Vanderbilt University. It is a configurable toolkit for creating domain-specific modeling and program synthesis environments [5]. GRL is one of such domain-specific modeling language. As described in the next chapter, this project built a GRL editor that generates GRL models based on GME. GRL models can be very complex and be used in large-scale software systems. GME supports hierarchy, multiple as-

pects, sets, references and explicit constraints. These are the fundamentals for building large-scale, complex models. This chapter briefly introduces GME features that are used in this project.

2.2.1 Basic Modeling Concepts

GME provides a set of basic modeling concepts that are used to create the domain specific modeling environments, which is generated from meta-models for these domains. These concepts can be described using the meta-meta-model shown in Figure 2. This model uses a class diagram to show the relationships among its concepts. It is considered as a model for creating meta-models, so it is also called *meta-meta-model*.

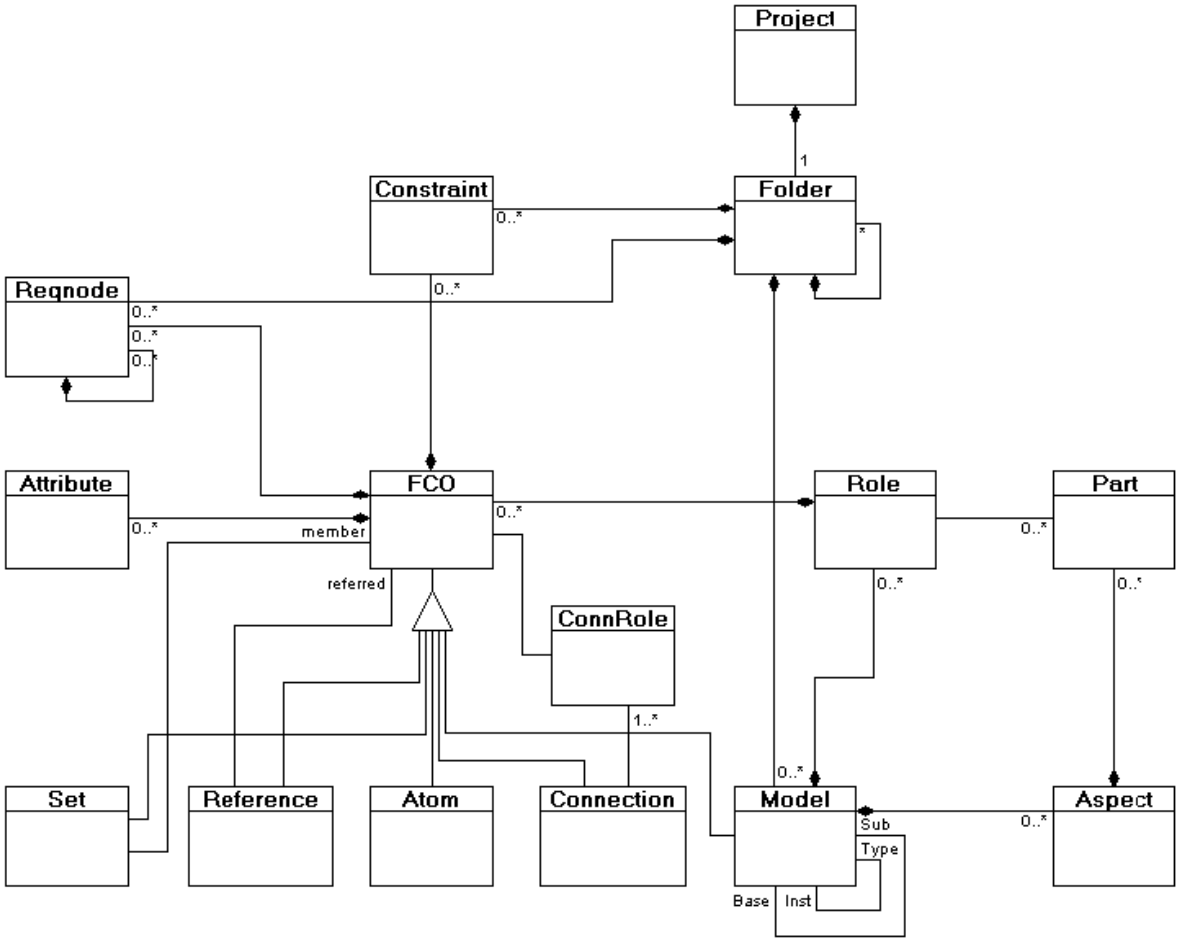


Figure 2 GME modeling concepts (meta-meta-model for GRL)

This figure shows that a GME Project contains multiple Folders, and a Folder contains multiple models.

There are a group of concepts called First Class Objects (FCOs) – Atoms, Models, Connections, References, and Sets. These objects share common attributes such as "Is Abstract". These FCOs are also called "parts". Since such an object can have attributes and relationships to other objects, they are very similar to the concept of "class" in UML, and thus the diagram in Figure 2 is similar to a UML class diagram. The following is a brief description of these concepts.

- *Atom* - An Atom is the elementary object in a GME model – it cannot contain parts. Each kind of Atom can have a predefined set of attributes. The attribute values are user changeable. An example of Atom in GRL editor is "Softgoal".
- *Model* - A Model is a compound object. It can contain other parts, and thus inner structures (models). This containment relationship creates a hierarchical decomposition of models. The depth of the hierarchy can be unlimited. Any object in this hierarchy must have at most one parent and this parent is a Model. The root of this hierarchy is called "root model". Notice that each part in a model must have a Role. How the parts construct a model and what are their roles are specified by the modeling paradigm. For example, GRLModel in a GRL editor (see Figure 12), is such a Model object.
- *Connection* - A Connection defines the relationship of two objects that are in the same model. It can be directed or undirected. It also has multiple Attributes. The modeling paradigm specifies what kind of connection can be used to connect what kind of objects in the model. A connection can be restricted by explicit constraints such as multiplicity. GME supports standard OCL in specifying constraints, as well as a group of its own predefined OCL types.
- *Reference* - A Reference always refers to a real object. That object can be in a different model. Any object except Connection can be referred. This is a very important feature for a large-scale complex model in which one object may need to be reused in multiple places – defined in one model but used in other models.
- *Set* - A Set is a more generic form of Reference. It specifies the relationship among a group of objects. All the members of a set must have the same parent.

Besides the above FCO objects, a model contains multiple Aspects.

- *Aspect* - An Aspect provides some kind of visibility control. GME has a predefined set of Aspects for each model. Every part can be assigned with certain Aspect as its primary Aspect, only in which it can be created or deleted. A mapping can be specified about what Aspect of a part can be shown in what Aspect of a parent model. For example, all the Attribute objects are created in the "Attributes" Aspect of the model.

Any FCO object can have multiple Attributes.

- *Attribute* - An Attribute is a property for an object. It has a name, an attribute of Type and value. The value of a Type can be "number", "string" or "enum". In the modeling paradigm, an attribute is defined to be a property for certain types of objects. OCL can be used to specify constraints on the use of the attribute. All Attribute objects are created in the "Attributes" Aspect of the model.

2.2.2 Type Inheritance and Model Library

Type inheritance is a very important feature for supporting a complex modeling. It is similar to a class inheritance in Object-oriented design. This concept is related to a model object. By default, a model created from scratch is a type. It can be sub-typed with new parts being added to the sub-types. Notice that parts can not be renamed or deleted in a sub-type.

A Model Library is in fact a GME project. It can be reused via importing into a project as a library, usually in the form of sub-types or new instances.

These important features are unique in GME, compared with other existing GRL supporting tools such as OME3 from the University of Toronto [23]. Type inheritance makes it possible for models to be reused and evolved in a development process. Chapter 6 will give further discussions on the extensibility of the GRL editor, which is a meta-model for the GRL models.

2.2.3 Decorators

GME supports the visual drawing of an object with a COM object called "decorator". This mechanism enables the creation of domain-specific modeling environment in which

customized visual representation of a concept can be created. All GME objects, except Connection, can be associated with a customized decorator.

This COM Decorator implements the interface IMgaDecorator (see document [2] for more information), which has public methods, such as Draw(), for the user to implement a specific type of drawing. In Chapter 4, such a customized Decorator will be discussed.

2.2.4 The Modeling Paradigm and the Meta-models

For GME to create an environment that will in turn create domain-specific models, a meta-model for this environment must first be set up. Meta-models can be defined with UML-like class diagrams in GME, like those in Figure 2. Using a meta-GME interpreter, GME translates this meta-model into the above mentioned modeling environment, also called a modeling paradigm. The Modeling Paradigm specifies what concepts are involved in a model, how these concepts are organized to make up a model, and what the relationships between the concepts are.

A meta-model is in fact a model that specifies how other models can be built. In this project, the GRL editor is one such modeling environments. Notice that in GME one model can be used to create a modeling environment for creating other models.

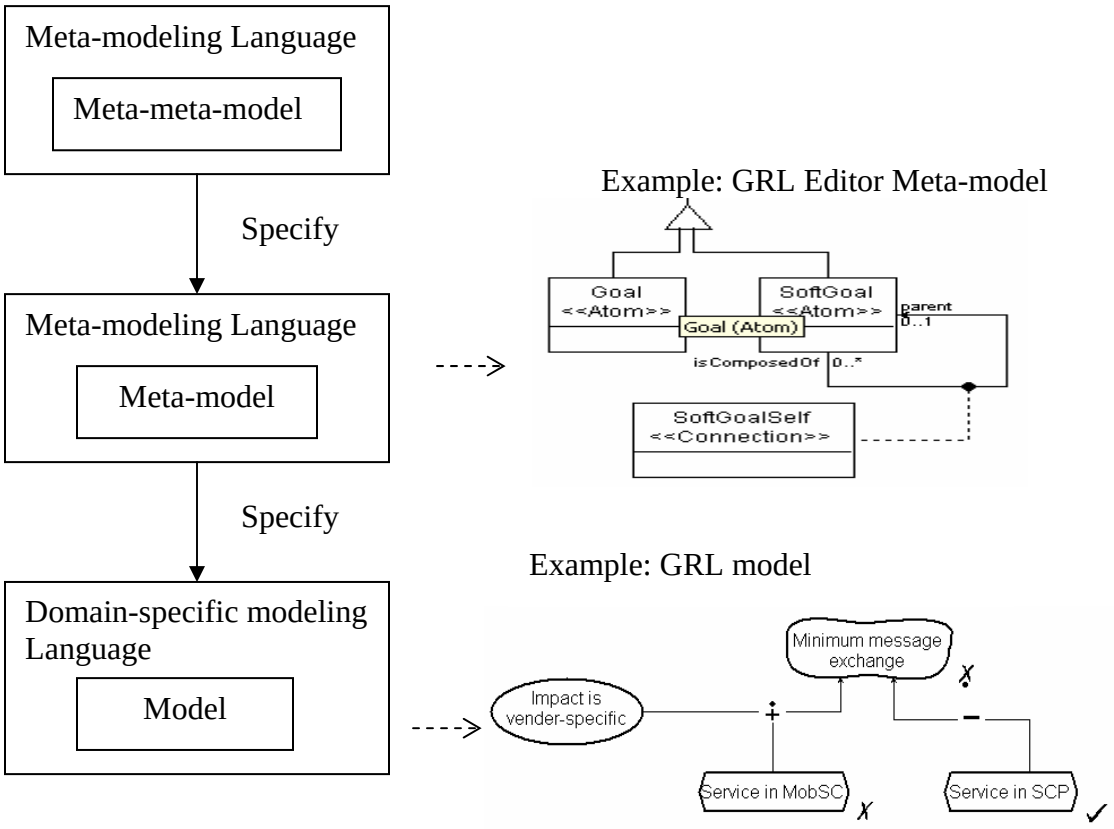


Figure 3 Layers of modeling concepts in GME

2.2.5 Creating Models using the GME User Interface

GME has a very easy-to-use user interface, as shown in Figure 4.

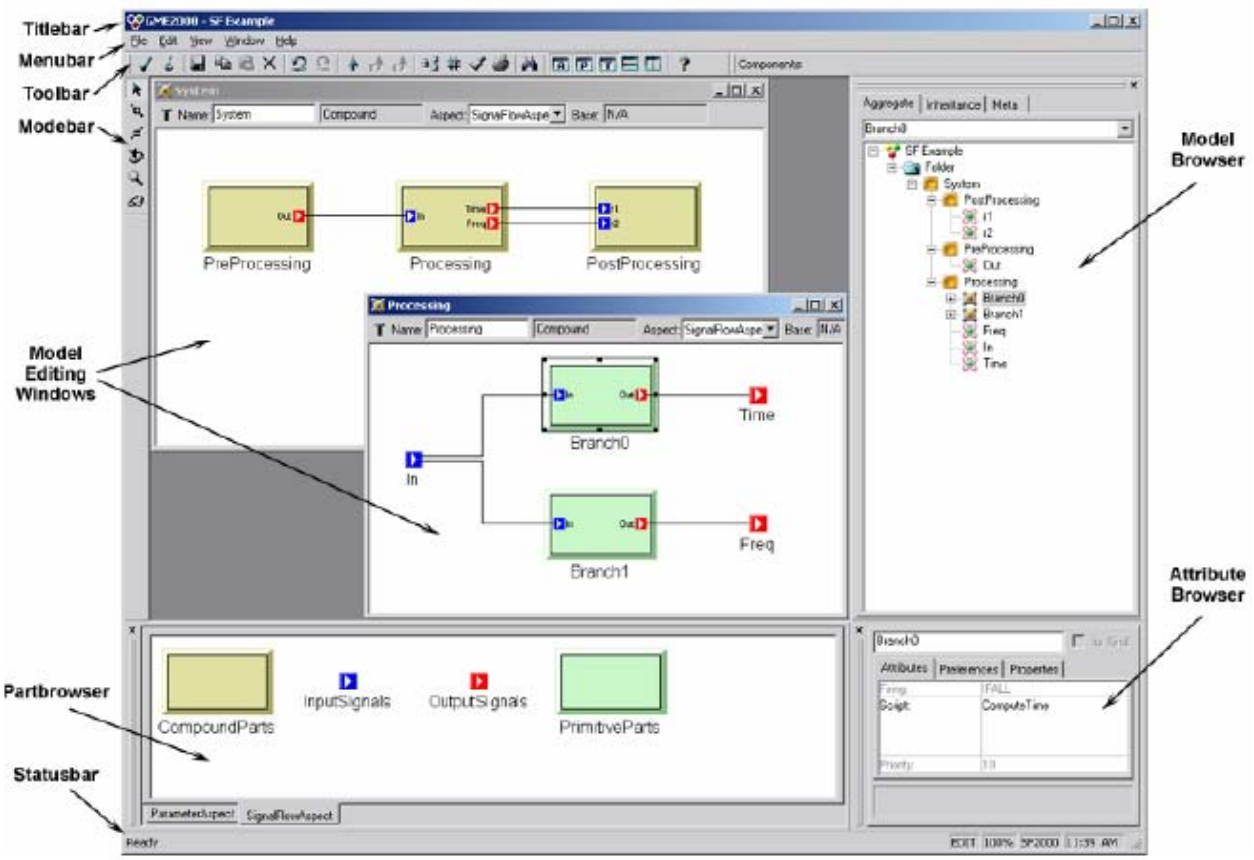


Figure 4 GME Main Editing window

The *Part Browser* is the window that holds all available parts (objects) that can be created in the current model and current aspect, e.g. CompoundPart and PrimitivePart in the above example. The *Model Browser* is a convenient tool for navigating through the model hierarchy in the current project, while the *Model Editor* window is the place where the model is constructed. The *Attribute Browser* is used to modify the attributes and preferences for the currently selected object.

The following shows the steps required from setting up a domain-specific modeling environment to generating a model in this new environment.

1. **Create the meta-model**. When creating the new project, specify a paradigm (that corresponds to meta-meta-model) based on which the new model (actually the meta-model) will be built. After the new project is created, it can be saved as a MGA file in binary format. Alternatively, it can be exported to a file in XML

format and be imported later. The base paradigm that is used in this project is the meta-GME paradigm "*MetaGME*".

In the new project, create objects that represent the domain-specific concepts that will be used to construct the domain-specific models. Define constraints, attributes and other elements such as aspects. The result of this step is the meta-model for constructing the domain-specific models.

2. **Interpret the meta-model and install the new paradigm**. The meta-model cannot be used directly to create domain-specific models. It needs to be interpreted by the interpreters associated with the paradigm based on which this meta-model was built. The result of this interpretation is a new paradigm that is the modeling environment for the specific domain. Since the "MetaGME" is used in constructing the meta-model, the interpreter should be the Meta-interpreter of GME.
3. **Register the paradigm**. The output of the model interpretation, i.e. the new paradigm definition file, is usually saved in an ".mta" file. Using the GME UI, this file can be registered on any user's machine into the Windows registry. After this process, the new paradigm is ready for use.
4. **Create new domain-specific models**. This is the time when the desired domain-specific models get created, in the newly created modeling environment (the new paradigm created in the previous steps).

2.2.6 Managing Paradigms

A meta-model can evolve in the development process. This poses an issue of paradigm backward compatibility – can an existing model be opened with a paradigm that has been updated?

If the paradigm evolves after it is used to create a certain model, it will be registered again while the older version of compiled paradigm still remains in the registry. This makes it possible for a model to be opened after its paradigm has evolved so drastically that it can no longer be opened with the new paradigm. However, in general, the paradigm will be backward compatible if the new paradigm does not delete or rename a part that has been used in an existing model.

If it does happens that a model cannot be opened with an updated version of paradigm, one solution is to use the old version paradigm to open this model, export it in XML format, and then import the XML format model in the new paradigm.

2.2.7 High-Level Component Interface

GME supports program synthesis, or model interpretation, through two interfaces:

- *COM Interface* - This COM interface provides access to the objects directly in a model. The properties of the objects can be modified programmatically by any language that supports COM. In GME, the COM interface is provided by two components: MGA and Meta. The MGA component is the main subject when a user wants to program with a GME model via the MGA object model.
- *Builder Object Network* - This is a higher-level C++ interface built on top of COM. It wraps many low level details with its wrapper classes and the programming framework, thus making the coding of model objects much easier. This is a good tool for generating any future interpreter for evaluating GRL models.

Chapter 3. GRL Editor Design

Our GRL Editor is a modeling environment. In GME, a paradigm defines such an environment. A paradigm is a GME interpretation of a meta-model. This chapter starts with the creation of a GRL meta-model and then introduces the whole design process of the GRL Editor with a tutorial. The GME environment used in this project is GME 4 [5].

3.1. Defining the meta-model

In order to create a GRL meta-model in GME, a definition of this model is needed. As discussed in section 2.1.3, a UML class diagram is used to define the GRL meta-model. Figure 5 is the temporary version of GRL meta-model definition as a UML class diagram.

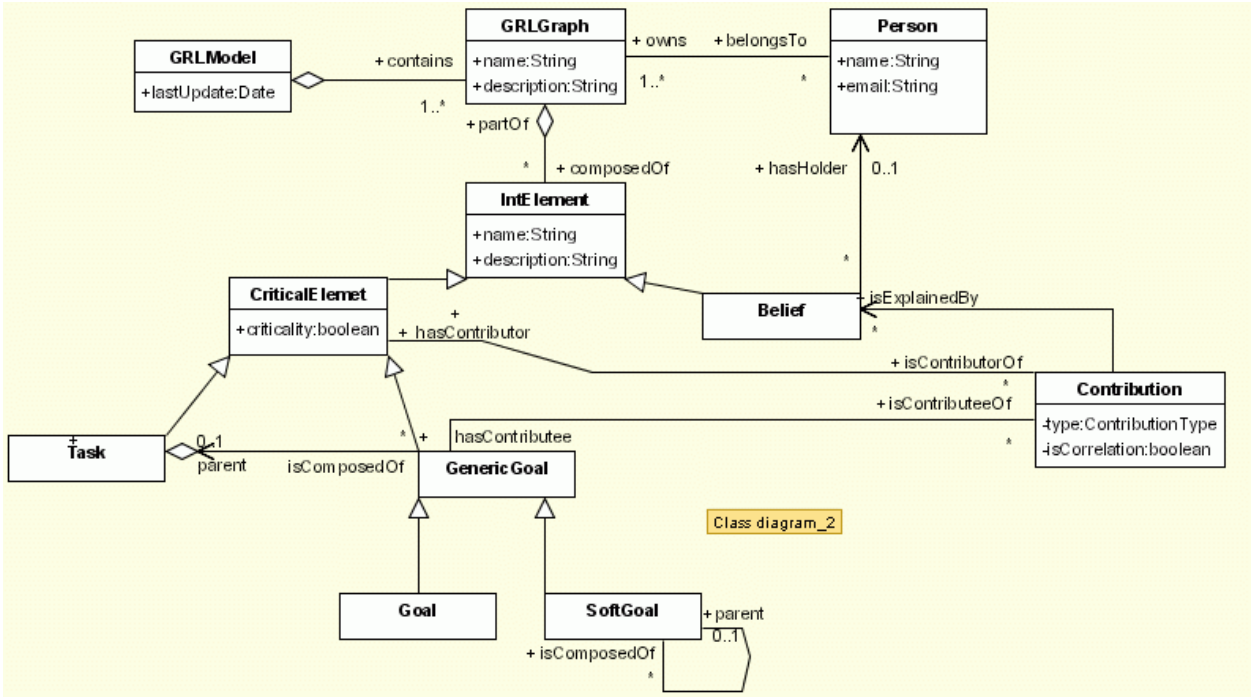
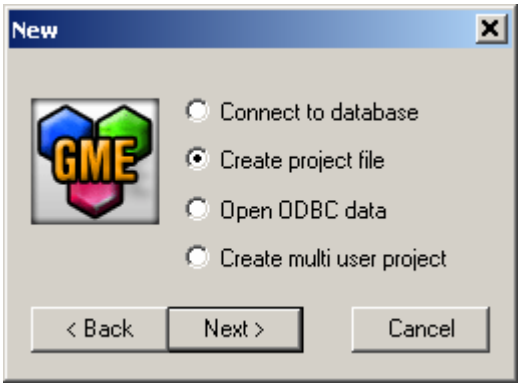


Figure 5 Simplified UML class diagram for GRL meta-model

3.2. Creating the Meta-model in GME

3.2.1 Creating a New Project

- From the menu item “File”, select the “New Project” item. A dialog box “Select Paradigm” is displayed. Select “MetaGME” in the available paradigms list (these are the paradigms registered for use by the current environment), then press the “Create New” button. The following dialog box is displayed.



- Choose “Create project file” and press “Next” button. In the following “Open” dialog, enter a name for this project (e.g., “MetaGRL”) and then press the “Open” button. An empty GME main window is displayed as shown in Figure 6. The components in this window are explained in Figure 4 and section 2.2.5.
- In the top-right “Model Browser” window, click on the “Root Folder” item and change its name to “MetaGRL”. This sets the name for the future paradigm that will be generated from this project. Since this project holds a model – the GRL meta-model – in the following part of this report, the terms “project” and “model” will be used interchangeably.

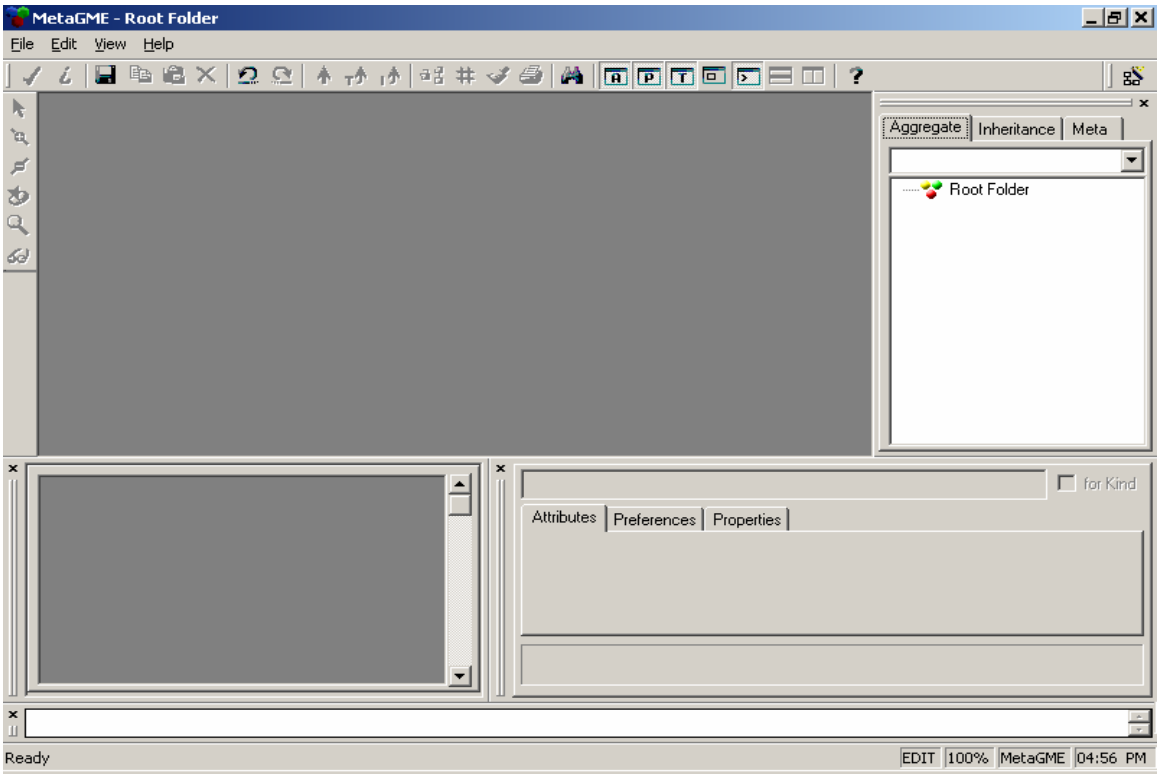
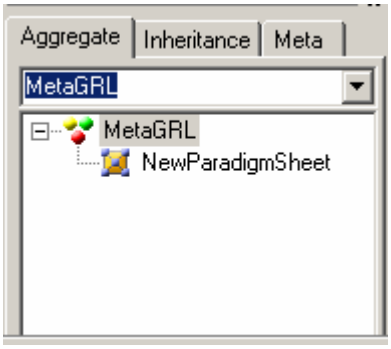


Figure 6 Empty GME Main Editing window

3.2.2 Insert a Model into the project

- Right-click on the “MetaGRL” item in the “Model Browser” window, and select the menu item “Insert Model” in the popup context menu. The only available sub-menu item is “ParadigmSheet”. Select this item and a model in this project is created, which is shown as a tree node under the root node “MetaGRL” in the following figure.



- In this window, click on the item “NewParadigmSheet” and give it a new name “GRLMetaModel1”. This is the first model (the meta-model for GRL editor) created in this tutorial.

3.2.3 Creating GRL Modeling Concepts in the Meta-model

Double-clicking on the tree item “GRLMetaModel1” in the “Model Browser” window will bring up the other editing windows for this model, as shown in Figure 7.

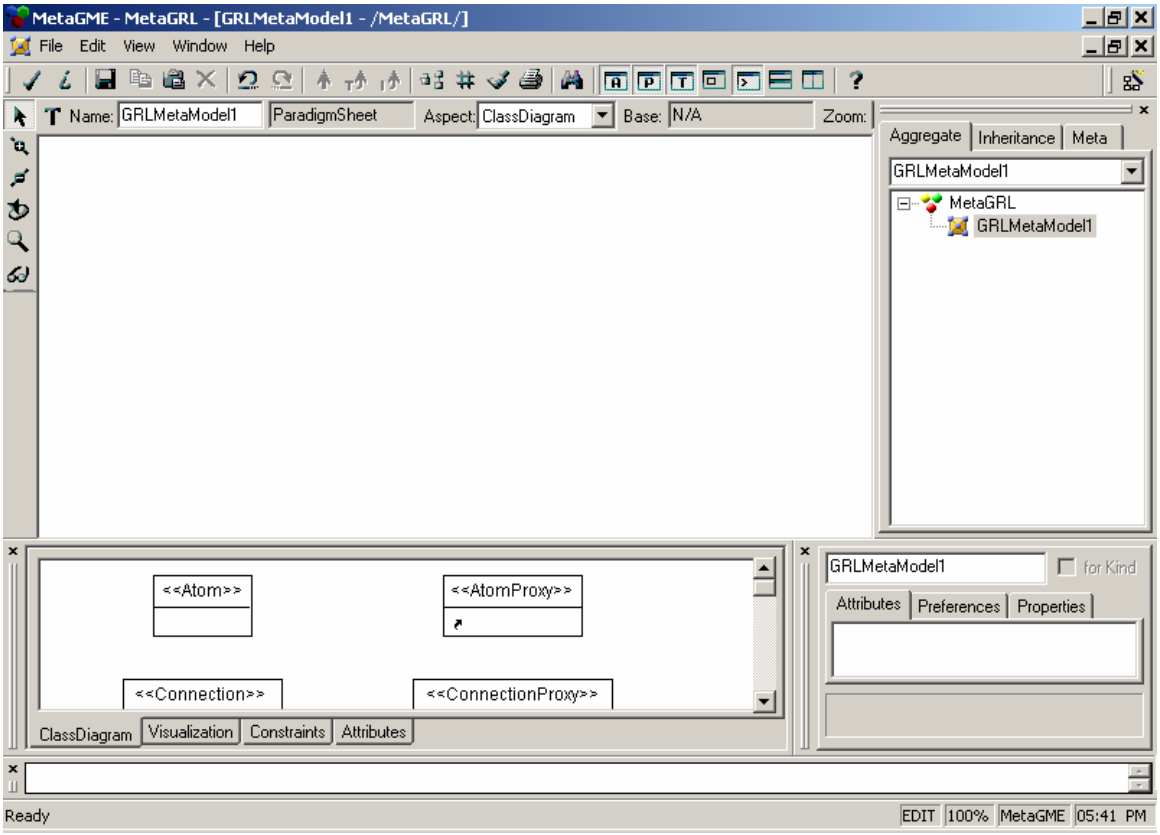


Figure 7 Editing windows with a blank model

Notice the current “Model Editing” window is blank. There are four types of model editing windows; each is called an “Aspect” of the model. The current one – “ClassDiagram”, is the default primary aspect of the model where objects can be added to or deleted from the model.

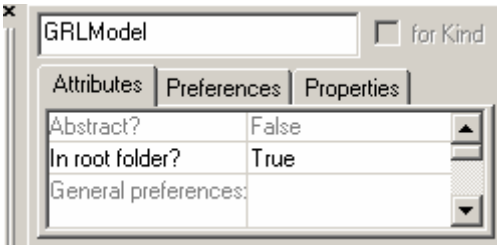
The “Parts Browser” window contains the icons for the available parts. Dragging one of such icon into the “Model Editing” window will create an object of the corresponding type. We will start creating this model from here.

As mentioned before, this model is based on the UML definition of the GRL Meta-model defined in Figure 5. Since the object "GRLModel" is the top-most object that will contain all the other elements in the model, it is thus the first one to be created.

Create a "Model" object for GRLModel

Such an object contains multiple "GRLGraph" objects. This indicates that it must be created with a "Model" type object in GME. To create a "Model" object, go to the "Parts Browser" window on the bottom-left part of the main window, locate the icon for "Model" object, drag and drop it into the "Model Editing" window.

Next, a name should be assigned to this Model object. This is done by clicking on this object and entering the name, i.e. "GRLModel", in the "Attribute Browser", as shown in the following figure.

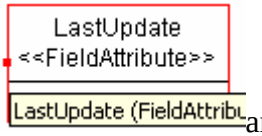


Notice that one important step to do is to set this object to be in the root folder of this project. This is done by going to the "Attribute" tab in the "Attribute Browser" window, select attribute "In root folder?" and set its value to "True". This will ensure that at least one part of the model is in the root folder of the project that is required by the meta-GME interpreter.

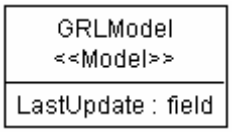
Create an Attribute for the GRLModel object

In Figure 5, we see that GRLModel class has an attribute "LastUpdate". Correspondingly, in the GME model, the GRLModel object should have an attribute for it. In GME, there are only three types of attributes – Boolean, enumeration and field. A Date data type in UML, just like a String type, can be implemented as a Field attribute in the GME model. The following shows how to do this.

1. Go to the "Attribute" aspect. In the "Parts Browser" window, drag "FieldAttribute" icon to the "Model Editing" window.
2. Enter a name "LastUpdate" for this Attribute object

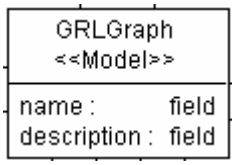
3. Click the  button to create a connection between the “GRLModel” and “LastUpdate” objects. Move the cursor above the “LastUpdate” object to show the red square like this 

and click. The first object selected is the contained object. Then move the cursor above the side of the “GRLModel” that is close to the “LastUpdate” object and click.
4. At this time, the connection is established. Go back to the “ClassDiagram” aspect; notice that the attribute of “LastUpdate” has been added to the “GRLModel” like the following.



Create a GRLGraph object

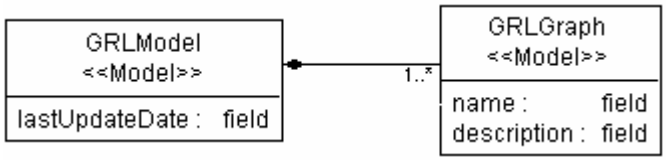
Since this object contains multiple IntentionalElements, it should be a Model type object in GME. Use similar steps described in the previous section, this object is created as the following.



Create a Containment connection between two objects in the model

Notice in the UML class diagram, there is a containment association between the GRLModel and GRLGraph objects. In GME, a Containment type of connection between two objects can be setup directly by using the button  .

The multiplicity of the connection is set by clicking on the “Attributes” tab in the “Attributes Browser” window, select the “Cardinality” attribute and set its value to “1..*”. The resulting model is shown in the following figure.



Create an Aspect in Visualization Aspect

The GME model must contain at least one Aspect. The above two entities cannot be accessed in a paradigm until they are contained in an Aspect entity in the meta-model. The following shows how to create an Aspect and use it.

1. Switch to Visualization Aspect of the Parts Browser window.
2. Drag an Aspect icon onto the paradigm window. Name the new item GRL1
3. Click button  and then right-click on GRL1
4. Left-click on each entity that is supposed to be visible in this one and only aspect.

Create Intentional Elements with Generalization

There is a class hierarchy for the Intentional Elements in the GRL meta-model, with the base class called "IntElement". In GME, this is done by dragging an "Inheritance" icon from the Part Browser into the editing window and then clicking button  on the modebar. The result is shown in Figure 8.

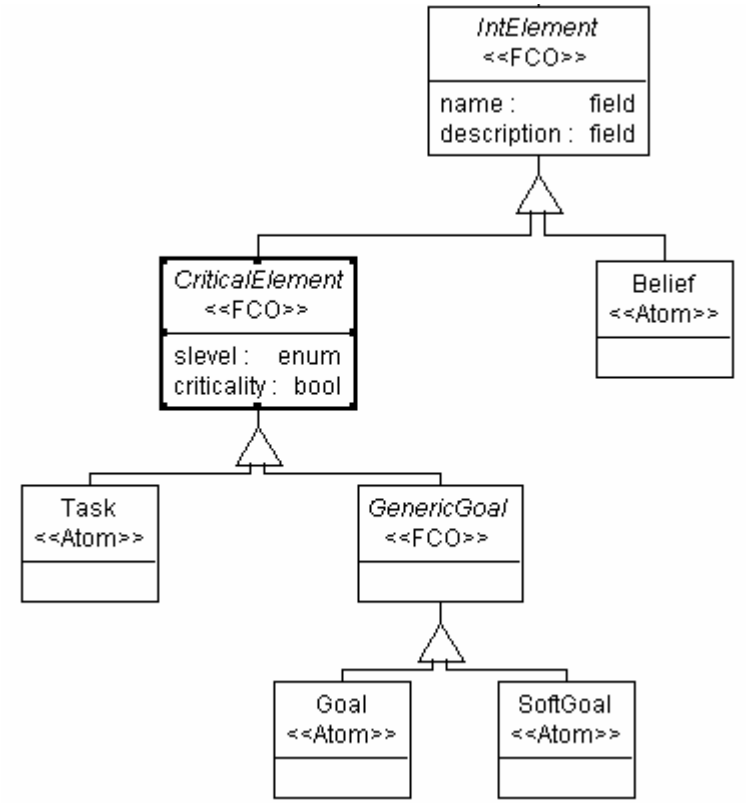


Figure 8 Intentional Elements class generalization

Notice that in the class hierarchy, only the leaf classes are of Atom type whereas the other classes are FCOs – only FCO can be a parent class for inheritance. Also, these

classes have attribute “Abstract” with default value as “True”. This means they cannot be directly created in a model. The root class is contained by the GRLGraph model class.

CriticalElement has an attribute “sLevel”, which corresponds to “Satisfaction Level” of a Goal/Softgoal or Task. It is an enumeration type; its value can be: Satisficed, Weakly Satisficed, Undecided, Weakly Denied, Denied, and Conflict.

Create an Association Relationship

In the UML diagram, there is an Association between Task and GenericGoal. This can be implemented in GME by a Connection object. An association relationship can be specified with attributes and constraints via a connection object. This makes GME powerful in expressing object relationships.

The following shows how to setup this relationship.

1. In the “ClassDiagram”, Part Browser, drag a “Connector” onto the editing window.
2. Drag a Connection object onto the editing window. Enter its name “TaskGoal”.
3. On the modebar, press button  .
4. First click on the right side of the Task object and then click on the Connector object. This creates a SourceToConnector relationship. Next, click on the right side of the Connector square and then click on the left side of the GenericGoal object to create a ConnectorToDestination relationship.
5. Click on the bottom of the Connector square. And then click the Connection object, in the popup dialog “Select Connection Role Type”, select “AssociationClass” and then press OK.
6. An Association relationship is established now between Task and GenericGoal (shown in Figure 9).

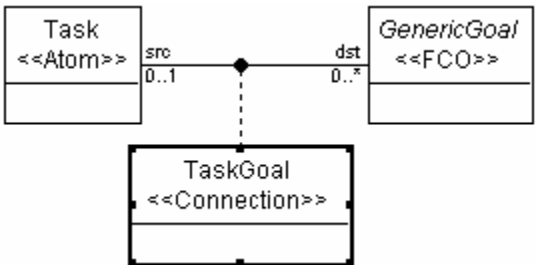


Figure 9 Association using Connection object

Containment of the Connection Object by the Model

In GME, an object is visible or accessible in the paradigm only when it is contained by its parent (model object) in the meta-model. If a connection is to be created between two objects in the GRLGraph paradigm using the button  , the Connection in the meta-model should be contained by the model class - GRLGraph. So, after the objects in Figure 9 are created, the TaskGoal object should be contained by GRLGraph.

Create Contribution and the Related Objects

In Figure 5, the relationship between CriticalElement and GenericGoal is connected by Contribution object. The Contribution object has an attribute called "Type", which is an enumeration type that has values such as "Make", "Break", etc. This object is implemented with an Atom type object with the same name in the meta-model. The following figure shows its relationship with other objects in the meta-model.

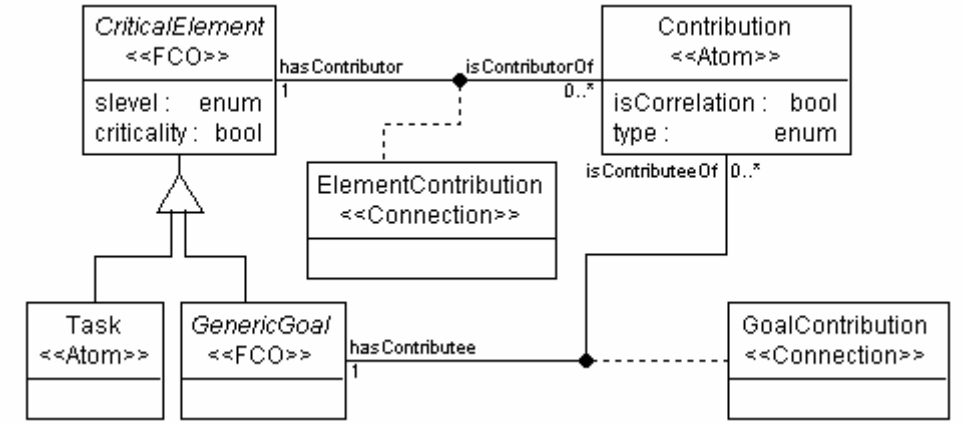


Figure 10 Contribution and its associated objects

Notice when creating the connection objects, the Rolename and Source/Destination primary attributes should be set correctly, as shown in the following table.

Source Object	**Destination Object**	**Rolename**	**Source/Destination primary attribute**
Critical Element	Contribution Source	Src	hasContributor
Contribution Source	Contribution	Dst	isContributorOf
Contribution	ContributionDestination	Src	isContributeeOf
ContributionDestination	GenericGoal	dst	hasContributee

Create a Reference for Person

In Figure 5, a Person may own multiple GRLGraphs. This object is at the same level in the model hierarchy as the GRLGraph. Person is also associated with a Belief, which is at a lower model level – it is contained by a GRLGraph. Any object is accessible only in a certain model level. For example, a Person cannot be accessed by a Belief directly since a Person object cannot be created within a GRLGraph, in which Belief and other lower level objects can be created and contained. For this reason, a Reference must be created in a GRLGraph, which refers to a Person object in the upper level that is contained in GRLModel. The following describes how to create such a reference.

1. From the Parts Browser window drag a Reference icon onto the paradigm (editing) window. Name the new item "PersonRef".
2. Connect the reference item to the source item "Person".
3. Contain the PersonRef item to GRLGraph.
4. Go to the Visualization aspect of the Parts Browser window and assign PersonRef to be contained in Aspect GRL1.

Figure 11 shows the created PersonRef item and its related entities. Notice that PersonRef is contained in GRLGraph, while Person is contained in GRLModel. GRLModel is an upper level model that contains multiple lower level GRLGraph models.

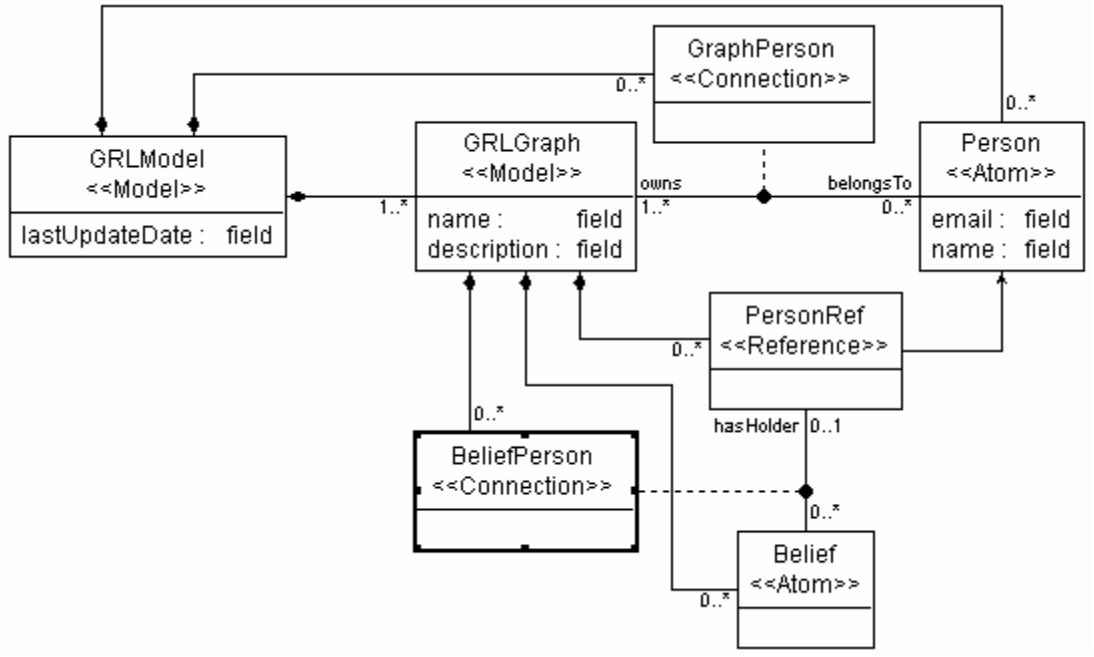


Figure 11 Person object and a Reference to Person

The Final Meta-model for GRL Editor

Now after all the entities have been created, a full version of meta-model for GRL Editor is created, as shown in the following figure.

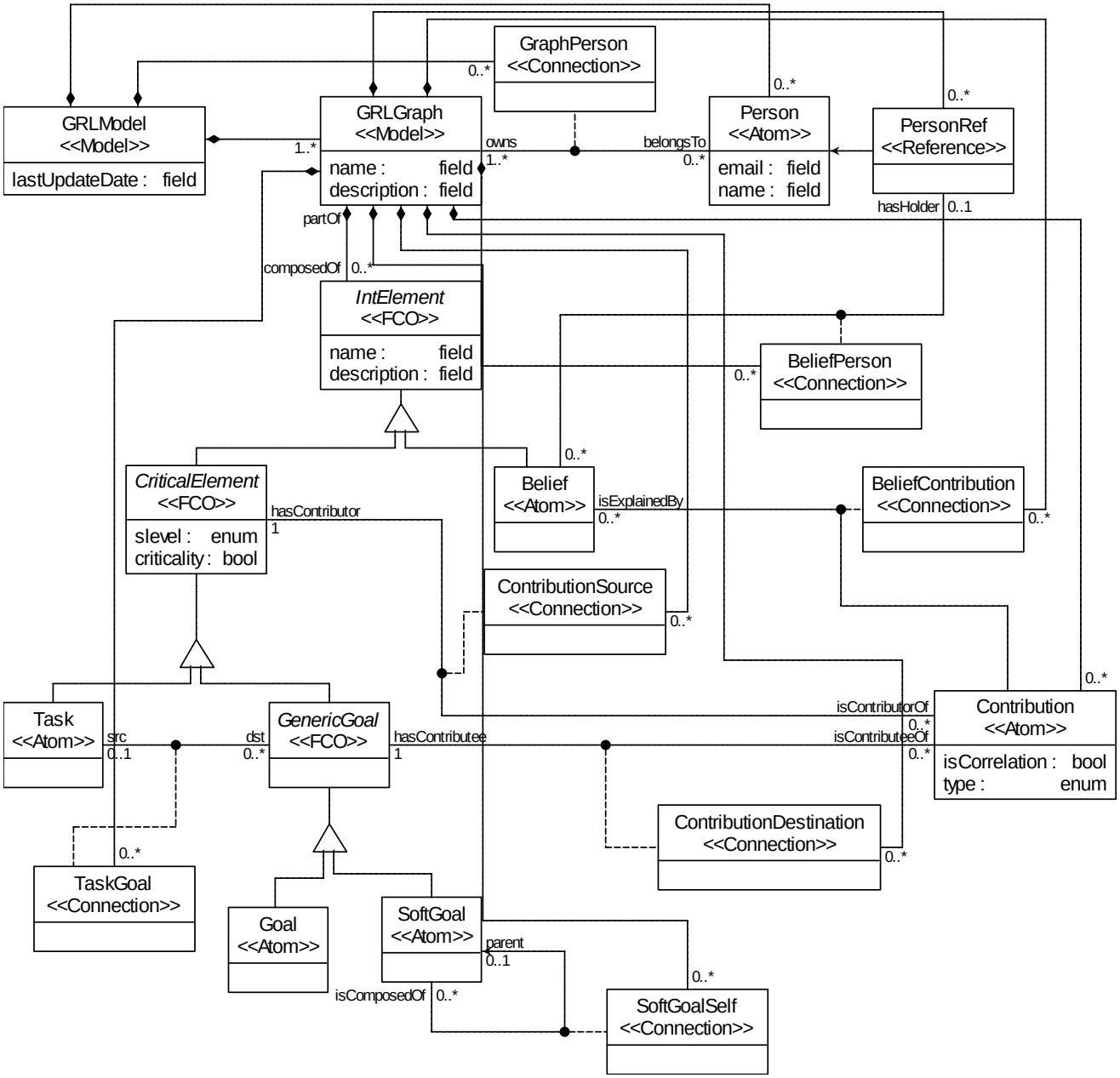


Figure 12 Meta-model for GRL Editor

Selecting Run Interpreter "MetaGME 2004 Interpreter" will compile this model into a paradigm – the GRL Editor. One can then create a new project and select the "GRLMeta2" paradigm; the new paradigm is ready for creating a GRL model.

Since the default display of an entity in a paradigm is a blue square shape as shown in Figure 20, b), the visual GRL model at this time looks plain with no symbolic shapes such as the ones shown in Figure 1. The following chapter will describe the decorator programming that helps enhancing the visual presentation of the model.

Chapter 4. Decorator Enhancement with COM Programming

The original meta-model can be compiled with an interpreter into a paradigm and then be registered. The registered paradigm within (working with) GME provides the modeling environment. The default visualization of the entities in the original paradigm is however unattractive – a blue square shape. This shape is the default shape for all entities, as shown in Figure 20, b). This chapter describes the COM programming done in this project to enhance the visualization features of our GRL editor.

4.1. Entity Visualization in GME

There are two ways to enhance the visualization of a GRL editor.

- Create specific icons and assign it to an entity in the paradigm
- Create specific drawing routine for a COM Decorator component and assign the Decorator to an entity.

Correspondingly, there are three alternatives for visualization enhancement of GRL Editor.

- Use customized icons for all entities.
- Use customized decorators for all entities.
- Use a combination of the customized icons and decorators.

In this project, the third alternative is chosen, for the reason that decorators provide much better effect, while icons are easier to implement. The following two figures show the actual icons and decorators in GRL Editor.

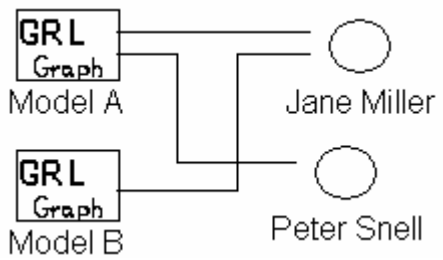


Figure 13 GRL model showing objects using icons

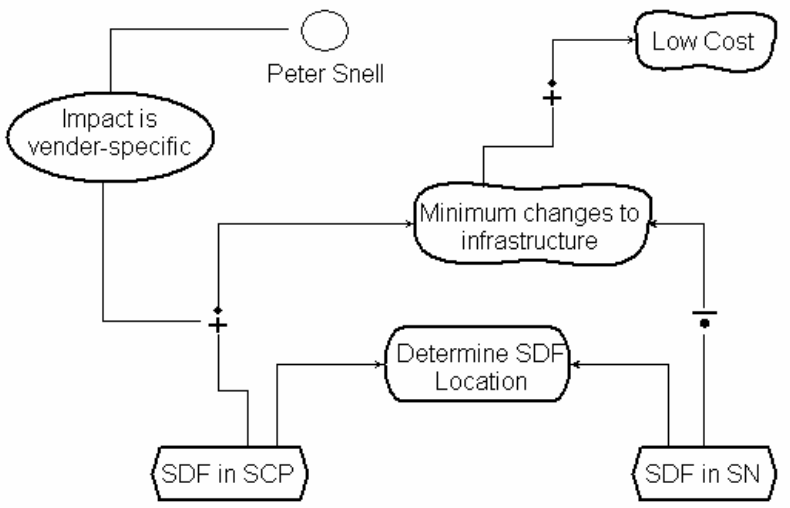


Figure 14 GRL model showing objects using Decorators

Creating a customized icon for an entity is simple – create a bitmap of the desired size and save in .bmp format; save the file in the "icons" subdirectory of the working directory where the paradigm DLL is located; in the meta-model, select the entity and go to Attribute tab of the property window, enter the icon file name to the field "Icon name".

The following section briefly describes decorator programming.

4.2. Decorator Development Kit and Programming

GME provides a development kit (framework) for programming with Decorators. Decorator for a certain entity can be specified in the meta-model. This is done by modifying the entity's Attribute "Decorator" to the program ID of the Decorator component, as shown in the following figure.

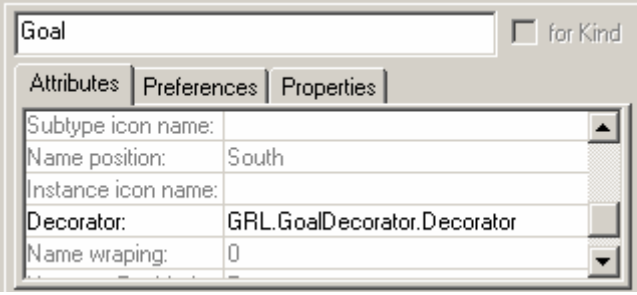


Figure 15 Modify Decorator attribute of the Goal entity

A Decorator is a COM component that implements interface *IMgaDecorator*. This interface provides methods such as *Draw*() for the customer. The decorator development kit provides a programming framework that includes default implementation of all the interface methods, and the necessary references to the MGA library model.

The decorator enhancement is done in the Decorator COM class *CDecorator* (which is placed in the file "*Decorator.cpp*"). A Decorator provides its service to a FCO object, such as a Task or SoftGoal entity. We call such a FCO object an associated object to the Decorator. The following describes the classes and the related methods that have been modified.

4.2.1 Decorators for Task, Goal, Softgoal and Belief

These Decorator classes caches 6 icon objects that correspond to the satisfaction levels. The *Initialization*() method loads the resources and initialize the icon objects, sets the data member *m_SatisfactionLevel* according to the associated FCO object's attribute "slevel". This method also calculates the size of the object region and save the result in *m_ShapeWidth* and *m_ShapeHeight*. Whenever the screen is refreshed, the *Draw*() method draws the shape of the object, with texts displayed within the shape, and with the proper icon shapes for satisfaction level of the object.

New data structure:

Enumeration SatisfactionLevel with values LevelNone, LevelSatisfied, …LevelConflict.

New class data members:

```
HICON m_IconSatisfied;      // Holding an icon object for a
                            // satisfaction level
```

```
HICON m_IconWeaklySatisfied;
HICON m_IconUndecided;
HICON m_IconWeaklyDenied;
HICON m_IconDenied;
HICON m_IconConflict;

SatisfactionLevel m_SatisfactionLevel;// Satisfaction level
                                      // of the object
int m_ShapeWidth;     // The width of the shape
int m_ShapeHeight;    // height of the shape
```

Modified class methods:

- *Initialize(IMgaProject *project, IMgaMetaPart *metaPart, IMgaFCO *obj)*

  This method is called by the GME whenever the associated FCO object is changed, e.g. attribute change. Notice that the parameter "obj" passes the reference to the associated FCO object to this Decorator class. The user can do further programming on the Decorator class based on the FCO's attribute values.

  The task of this method is to initialize the data members, including those introduced in this project – the icon handles, satisfaction level, and the shape width and height.

  Notice that the first statement (new code being added) in this method:

  AFX_MANAGE_STATE(AfxGetStaticModuleState( ));

  It is required since we have added codes to perform queries to the model database. These operations should be executed within a transaction.

- *GetPreferredSize(long* sizex, long* sizey)*

  This method is called by GME whenever it is going to redraw the decorator. The original method returns default constant values for the size of the shape. The modified code returns the two data members whose values are determined by the helper method *CalcShapeAndPositions*().

- *Draw*(*HDC hdc*)

  This method is called whenever the GME wants to draw the decorator.

  It draws the satisfaction level symbol with a corresponding type of icon. It then draws the shape of the object – e.g. a cloud shape for a Softgoal, a round rectangle shape for a Goal object. When drawing the cloud shape, the *PolyBezier*() method of the MFC drawing context class is used as it helps drawing curved lines.

This method then displays the name text of the FCO object. The texts are formatted into multiple lines according to the size of the shape and the length of the text, which is determined by the helper method *ParseDisplayStrings*().

New class methods:

- *Void CalcShapeAndPositions*(*CDC *pDC*) //(new method)
  Calculates the size of the decorator shape, depending on the length of the FCO object name text.
- *Void ParseDisplayStrings*(*CString str, CDisplayArray *display_strings*)
  Parses the FCO object name text (one line string) into a multiple line string array so that they can be displayed in the decorator shape in multiple lines.

4.2.2 Decorator for Contribution Entity

This decorator is different than the above mentioned one in that it uses different data types and displays a simpler type of shape – specific icon.

The decorator class initializes the data members that represent the type of contributions and the flag that represents a normal vs. correlation type of contribution. It also initializes the icon objects that will be used to draw the specific shape according to the type of the contribution. When the class is called to draw the decorator, it draws the icon according to the contribution type.

New data structure:

Enumeration ContributionType with values TypeBreak, TypeHurt…TypeEqual.

New class data members:

```
HICON m_IconBreak;    // A handle to a type of icon object
HICON m_IconHurt;
HICON m_IconSomeneg;
HICON m_IconUnknown;
HICON m_IconMake;
HICON m_IconHelp;
HICON m_IconSomeplus;
HICON m_IconEqual;
HICON m_IconContribution;
bool m_IsCorrelation;      // Type of the contribution –
                           // normal vs. correlation
ContributionType m_ContributionType;  //contribution type
```

Modified class methods:

- *Initialize*(*IMgaProject *project, IMgaMetaPart *metaPart, IMgaFCO *obj*)

  This method is called by the GME whenever the associated FCO object (the contribution object) is changed, e.g. attribute change.

  The task of this method is to initialize the data members, including those introduced in this project – the icon handles, the type of the contribution (normal/correlation) and the contribution types.

  Notice that the first statement (new code being added) in this method:

  AFX_MANAGE_STATE(*AfxGetStaticModuleState*( ));

  It is required since we have added codes to perform queries to the model database. These operations should be executed within a transaction.

- *Draw*(*HDC hdc*)

  This method is called whenever the GME wants to draw the decorator.

  It draws a rectangle shape if the contribution type is not specified (which is the case when the contribution object is created initially, or it is displayed in the Part Browser window.

  It will draw a specific icon ("+" for "Some+") if the contribution type is set.

4.3. Deployment of Decorator COM servers and Icons

There are 5 Decorator COM (servers) developed in this project, each with a separate Visual C++ project, for the meta-model entities Task, Goal, Softgoal, Belief, and Contribution. Their program ids are named in the following format.

GRL.<Entity Type>Decorator.Decorator

<*Entity Type*> is one of the above five meta-model entities.

The COM servers are compiled into corresponding DLLs. These DLLs can be deployed in any directory and registered in the Windows registry using the tool "regsvr32.exe".

When registering the COM servers, if certain dependent DLL (on which the COM server is dependent) does not exist, the operation will fail. In this case, a free downloadable Microsoft tool application "Depends.exe" can be used to identify which dependent DLL is missing, download this DLL and then register the COM server again.

The icons for the meta-model should be put in the sub-directory “icons” relative to the directory where the meta-model locates.

Chapter 5. Creating Models with the GRL Editor

Our GRL Editor has two layers, or levels, of modeling entities. The top level contains only two entities – GRL Graph and Person. A GRL Graph contains a number of other entities that are located in the second level of the model. When a new model is created, the user can only create top level entities first. Then, via double-clicking on a Graph object, a model in the second level can be created.

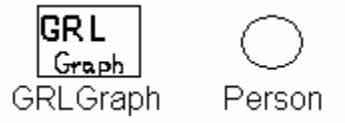


Figure 16 Top level entities in GRL Editor

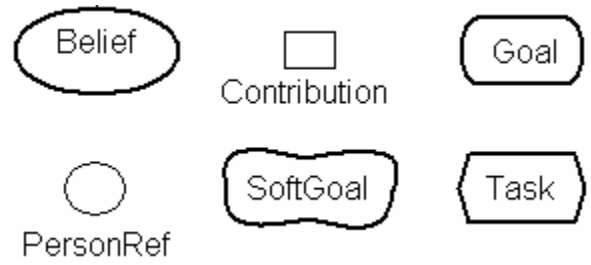


Figure 17 Second level entities in GRL Editor

The following shows how to create a sample GRL model.

1. Create a new project using menu item File/new project. Choose paradigm "GRLMeta2" (which is created from the GRL meta-model "GRL2.mga").
2. In the Model Browser window, right-click on the "root folder" node and select GRLModel from the "Insert Model" menu item.
3. Double-click on the created "NewGRLModel" node, drag "GRLGraph" icon from the Part Browser window into the Editing window. Click on the created object and go to Attribute tab of the property window to enter a name for the object, e.g. "GRL Model1".

4. Drag to create a Person object and give the name “Peter Snell”.
5. Create a connection between these two objects. The model is shown in the following.

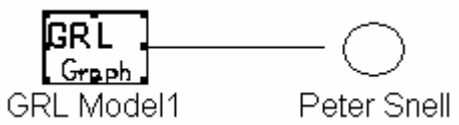


Figure 18 Model objects in the top level

6. Double-click on GRL Graph1 object to create the second level model entities.
7. Drag a Task icon onto the editing window and name it “SDF in SCP”.
8. Create Task “SDF in SN” and Goal “Determine SDF Location”.
9. Create connections from the Task to the Goal objects. For better visual effect, click on the connection and go to property window, preferences tab, attribute “destination end style”, select “arrow”.
10. Create SoftGoal “Minimum changes to infrastructure”.
11. Drag to create a Contribution. Put it between Task “SDF in SCP” and the SoftGoal objects. Click on the Contribution object, go to its “Type” attribute and set it to “Help”.
12. Create connections from Task to Contribution object and from Contribution to SoftGoal objects. Create a “Hurt” Contribution between the other Task object and the SoftGoal.
13. Create another SoftGoal “Low Cost” and a Contribution from the previous SoftGoal
14. Create a Belief and connect it to the Contribution created in step 13.
15. Create a PersonRef object. Create the reference connection to its source by clicking on the “Peter Snell” node in the Model Browser window and drag onto the PersonRef object and drop. Optionally give this object a name, e.g. “Peter S”.
16. Create a connection between the PersonRef and Belief objects.

By now, the model is complete. The next important step is to check the validity of this model against the OCL and multiplicity constraints (i.e., constraint evaluation). To do this, right-clicking on the model node in the Model Browser window and select the con-

text menu item “Constraint/Check All”. GME will inform you of the entities that violate constraint conditions, although the information is somewhat hard to be used to identify the problematic entity if the model is very complex. A good practice is to go this check frequently during the model development process.

Although model evaluation is out of the scope of this project, it has been made possible with the decorator programming. In the decorators for the Intentional Elements, there is an attribute “sLevel”, which can be set to one of the six satisfaction levels. The following shows how to set the evaluation result to this model.

1. Set the Satisfaction Level of the Task “SDF in SCP” to “Satisficed” via Attribute “sLevel”. Set the other Task to “Denied”.
2. Set the Satisfaction Level of the Goal object to “Satisficed”.
3. Set the two SoftGoal object to Satisfaction Level “Weakly Satisficed”.

The final model is shown in the following figure.

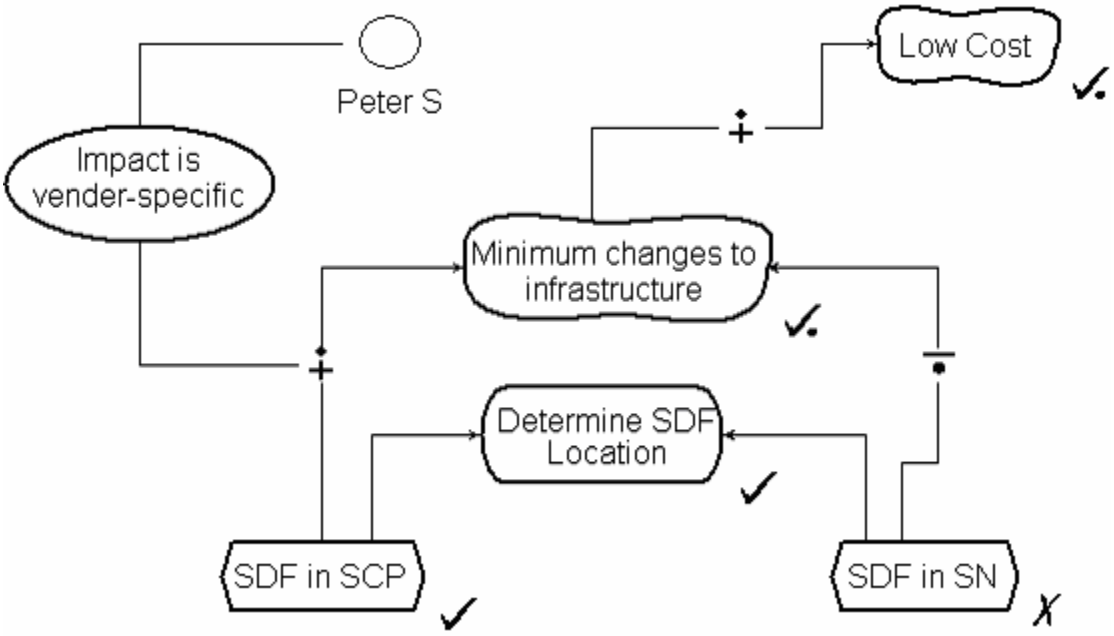


Figure 19 Sample GRL model

Chapter 6. Meta-model Evolution Experiments

GRL is a new and evolving notation. Its meta-model will be updated and extended in the future. One important aspect of a GRL editor is the capability of easily maintain and extend older models with the upgraded versions of the editor. In this report, we call this feature *backward compatibility* of the GRL editor. GME supports a mechanism that resolves the issues of upgrading a modeling environment (as discussed in section 2.2). In this chapter, we will perform some experiments to explore the real extensibility of the GRL editor.

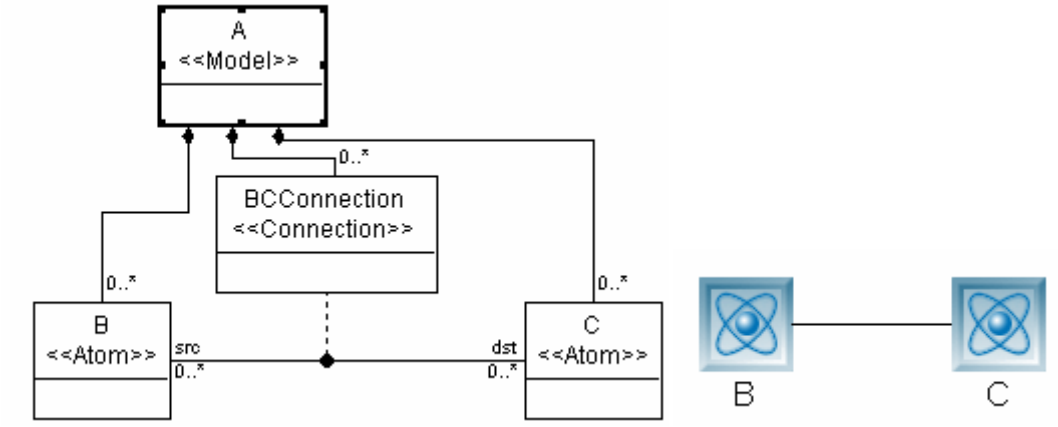
A GRL meta-model is made up of intentional elements, links between elements, references to elements, and attributes of the elements and links. In each of the following experiments, the study will focus on the change to one of these GRL meta-model entities.

6.1. Adding or Deleting an Element and Link

6.1.1 Adding a New Element and Link

Experiment No. 1

Suppose the meta-model contains entities A, B, and C, as shown in Figure 20, a). Figure 20, b) shows a model that is created from this meta-model. The purpose of this experiment is to test when new entities are added to meta-model, how the upgraded paradigm behaves.



a) Meta-model b) Original Model

Figure 20 Meta-model and model for Experiment No.1

- **Step 1:** Add new element and link. Figure 21 shows a new meta-model, in which a new entity D is added, along with the new association link between C and D.

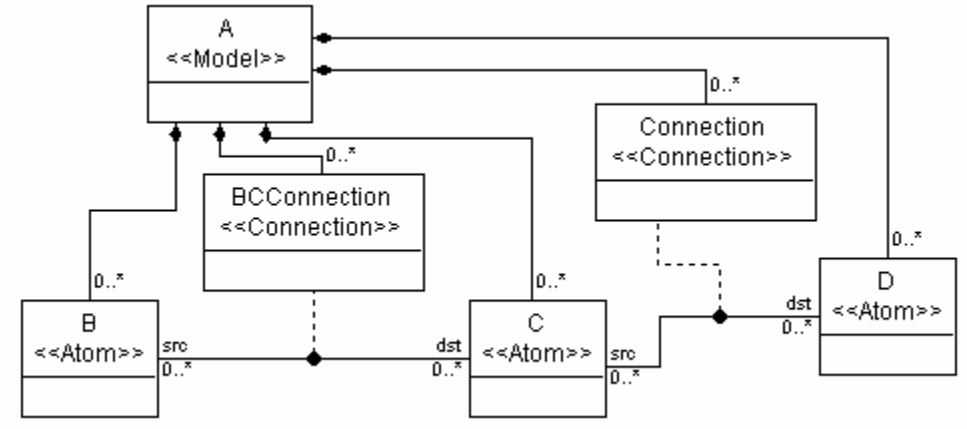


Figure 21 Meta-model with a new entity D and new link

- **Step 2:** Compile this meta-model and create an upgraded GME paradigm. And then open the previously generated model shown in Figure 20, b). The following warning message is displayed by GME.

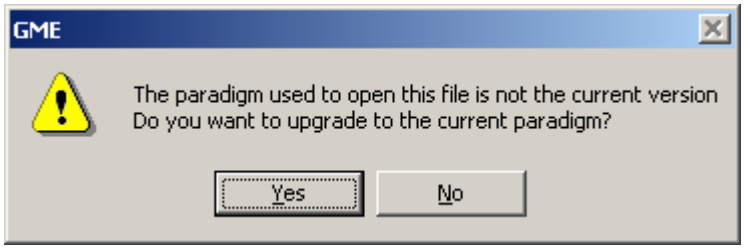


Figure 22 GME warning message when paradigm is upgraded

Press “Yes” button to try open the model with an upgraded version of paradigm. The model is opened correctly. The new entity (D) is able to be inserted into the model as shown in the following.

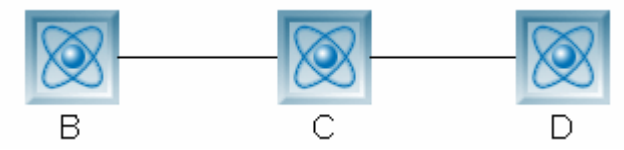


Figure 23 Model with new element D

Conclusion: Adding a new Element and Link to a meta-model will not break the backward compatibility.

6.1.2 Rename a Used Link

Experiment No.2

Suppose the original meta-model is as shown in Figure 21, where the connection entity between C and D is called “Connection”. The target model is as shown in Figure 23, in which the connection between C and D exists. Export the target model before the experiment. The purpose of this experiment is to test how the editor would behave when one of the links used in the meta-model is renamed.

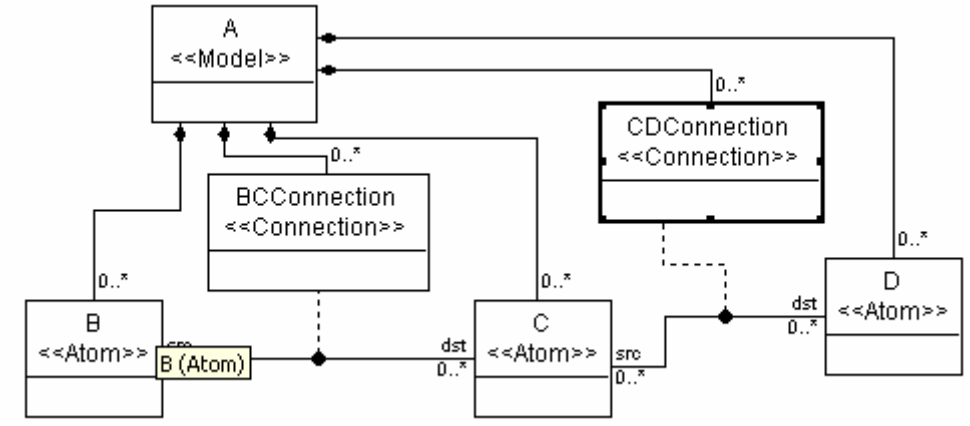
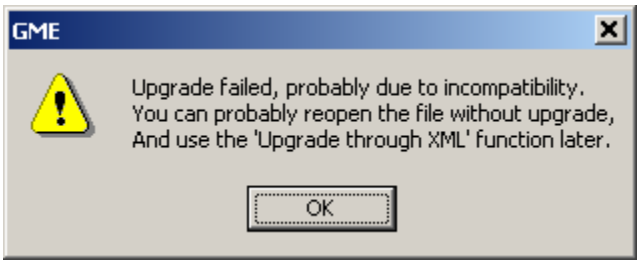


Figure 24 Meta-model with one link renamed

- **Step 1:** Open the original meta-model and rename the entity “Connection” to “CDConnection”.
- **Step 2:** Compile this model and then open the target model. The warning message of upgrading the paradigm is displayed.

- **Step 3:** Press “Yes” to the dialog and the following error message is displayed.

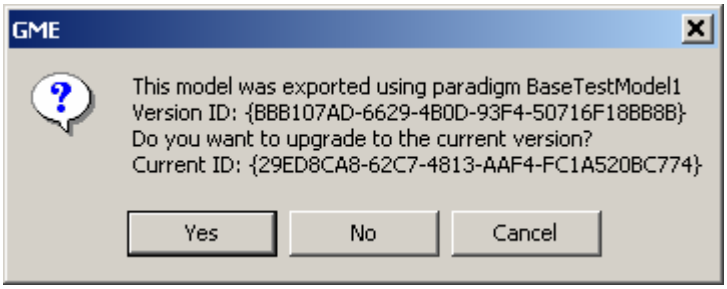


Press “OK” displays the error message “Could not open project: Meta incompatibility”.

Conclusion: Renaming a used link will break the backward compatibility.

Sub-Experiment: using XML export of the model.

- **Step 4:** Import the target model via menu item “File/Import XML”. Press “Open” to the displayed “Open” dialog. The following message is displayed:



- **Step 5:** Press “Yes” to the above dialog and displays the error message of “Error importing XML file”.

Conclusion: Exported XML format model cannot be imported after the meta-model changes the name of a used link.

Notice that in some cases, using an exported XML model can recover a model that can no longer be opened with a .mta file, after the paradigm is upgraded.

6.1.3 Rename an Unused Link

Experiment No.3

Suppose the original meta-model is as shown in Figure 21, where the connection entity between C and D is called “Connection”. The target model is as shown in Figure 20, b). Figure 23, in which the connection between C and D is not used. The purpose of this ex-

periment is to test when one of the unused links in meta-model is renamed, how the editor would behave.

- **Step 1:** Rename the “Connection” to “CDConnection” in the meta-model
- **Step 2:** Compile the meta-model and then open the target mode.

Conclusion: The model is opened successfully.

6.1.4 Deleting a Link and an Element

Experiment No.4

Suppose the existing meta-model contains A, B, C, and D, as shown in Figure 21. And the model created from the corresponding paradigm is shown in Figure 25. Notice in the target model, entity D and its association is not used. The purpose of this experiment is to test how the upgraded paradigm behaves when an unused link and element is deleted in the meta-model.

Figure 25 Original model for Experiment No.4

- **Step 1:** Delete link (“Connection”) and element (D) in the meta-model. The resulting meta-model is the same as Figure 20, a). Open the original meta-model and rename the entity “Connection” to “CDConnection”.
- **Step 2:** Compile this model and then open the target model. The upgrade current paradigm warning message is displayed.
- **Step 3:** Press “Yes” to the dialog and the model is successfully opened.

Conclusion: Deleting an unused link or element does not break the meta-model’s backward compatibility.

Experiment No.5

Purpose: Deleting a used link and Atom entity and test the backward compatibility.

The meta-model used is as shown in Figure 21. Target model: shown in Figure 25.

Test steps: Delete D and the Connection entity between C and D in the meta-model, compile the meta-model and then open the target model.

Conclusion: A model cannot be opened after deleting a used link or element in the meta-model.

6.2. Adding or Deleting an Attribute

In the experiments in this section, the original meta-model is shown in Figure 26.

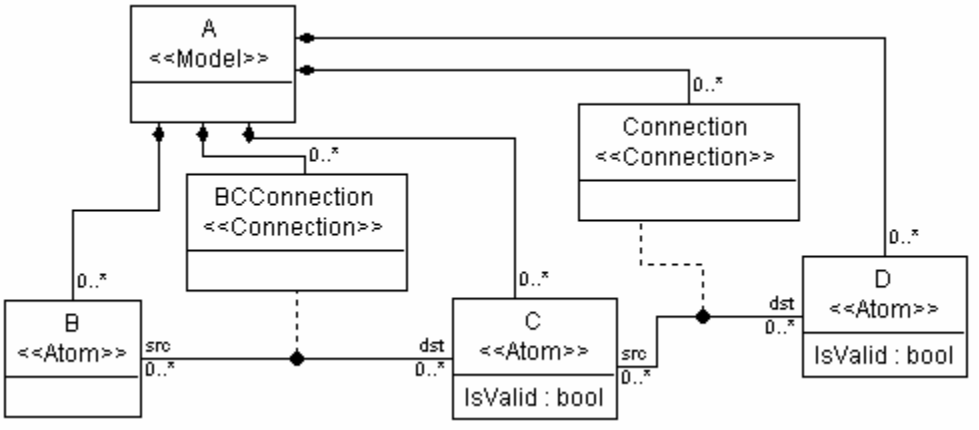


Figure 26 Meta-model for experiment with Attributes

The two target models are shown in the following figure.

a) Model I (without Element D) b) Model II (with Element D)

Figure 27 Target models for experiment with Attribute

The following is a table that lists the five experiments and the results.

Compatibility Experiment with Attribute			
No.	**Description**	**Target Model**	**Result**
1	Add a new Attribute of type Field to element B	I, II	Yes - Model I and II are opened correctly with the upgraded paradigm. New attribute is seen in object B with its default value
2	Rename “IsValid” Attribute in D to “IsNew”	II (D is used in this model)	Failed – the model cannot open.
3	Rename “IsValid” Attribute in D to “IsNew”	I (D not used in this model)	Yes
4	Delete “IsValid” Attribute in D	II	Yes – Model opened. The attribute is not present in object D
5	Delete “IsValid” Attribute in D	I	Yes

6.3. Adding or Deleting Reference

References are heavily used in a complex modeling system where one definition of an entity is used in multiple places and models. The following study the meta-model backward compatibility affected by changes to a reference.

The original meta-model is shown in the following figure.

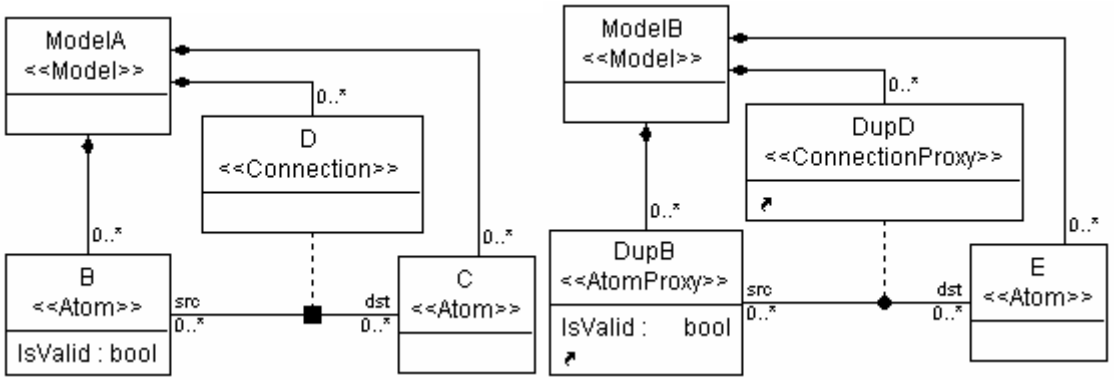


Figure 28 Original meta-model using references

The following is the target model that is created from the ModelB in the above meta-model. In this model, entity B is actually a reference that refers to entity B that is defined in ModelA.

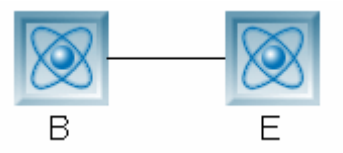


Figure 29 Target model for experimenting with reference

Notice that the "Parts Browser" window (showing in the following figure) for ModelB contains two types of elements – B and E, rather than DupB and E.

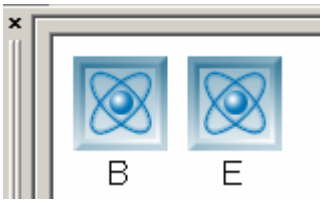


Figure 30 Parts Browser window of ModelB

The following is the table that lists the experiments and the results.

Compatibility Experiment with References		
No.	**Description**	**Result**
1	Rename the reference source B in model A to "NewB" in ModelA	Failed – Cannot open the target model
2	Rename the reference DupB to "NewDupB" in ModelB	Yes - the model opens.
3	Delete the reference DupB in ModelB (DupB is used in the target model)	Failed – cannot open the target model
4	Delete the reference source B in ModelA	Failed – Model cannot open

6.4. Conclusions

A meta-model evolves in the development process. An upgraded paradigm is backward compatible if the change in the meta-model does not rename or delete an element, link, reference and reference source that has been used to create an object in a model. With the exception of the Attribute – renaming a used Attribute will make the upgraded paradigm unable to open the model; but deleting a used Attribute does not break the backward compatibility of the paradigm. If adding any type of the above parts, or renaming/deleting any unused (not an object exists in the model) parts in the meta-model, the paradigm can be safely extended.

Chapter 7. Discussion

The project develops the GRL Editor based on a simplified version of a GRL meta-model. The GRL meta-model will evolve and thus the GRL Editor should be able to be extended in the future. This chapter briefly discusses improvements that can be made to the GRL editor, a comparison of GRL Editor with other works or products, and the future work on model evaluation.

7.1. Related Work

7.1.1 UML 2.0 Profiles

UML 2.0 uses profiles to extend the UML meta-model and customize it for a specific domain [9]. UML profiles are similar to paradigms in GME. There are two ways of using profiles:

- *Stereotype Mechanism* – Stereotypes that extend basic UML elements are used. Extensions include customizations of names, attributes and appearance. In this way, each GRL element can be implemented as a stereotype of a UML class. Although constructing a profile is rather simple, the created modeling environment still includes all the basic UML elements that were extended. In essence, this does not lead to a real domain-specific environment.
- *Meta-model Extension Mechanism* – In addition to the functionality of the previous type, this mechanism provides meta-model extensions of non-basic UML element, such as class diagrams, by extending the UML meta-model. GRL models can be represented as a meta-class extension of UML class diagrams. This mechanism is more powerful but is more complex to implement, with restrictions. However, the resulting environment can be restricted to a domain-specific language.

There are two software tools that are UML 2.0 compliant and thus can be used to implement a GRL modeling environment via the mechanisms described above.

IBM Rational Software Architecture (RA)

Rational Software Architecture is a design and development tool that leverages model-driven development with the UML for creating well-architected applications and services [5]. It is built on top of the Eclipse platform and thus has the powerful features that are provided via the Eclipse open source APIs. It only provides the stereotype mechanism profile. With regards to supporting development of custom meta-model, it has some limitations: GRL Actor boundary elements cannot be implemented; we cannot customize link styles; and we cannot customize the shape of an element directly (requires Eclipse-based Java API) [9].

Telelogic TAU G2

This tool supports model-driven software design using profiles of the two types of mechanisms mentioned previously [16] [9].

With stereotype extension profile, the functionality and limitations are similar to RA. For example, actor boundaries cannot be implemented.

Meta-model extension profile is however more powerful, with customizable diagram types and the modeling UI (shape of element). The limitations, such as Actor boundary support, are the same as with stereotypes.

The Pros and Cons

Compared with our GRL editor, these tools use profiles to customize a meta-model. The advantages are: easy to create a profile; access to Eclipse open source APIs and work with other Eclipse plug-ins; Java API programming is easier than COM programming. There are some issues with these tools when creating a GRL editor: domain specific meta-model in Stereotype mechanism is an extension of the basic UML elements, thus it is superfluous to most of the real needs of a domain specific modeling such as with GRL. Meta-model extension mechanism is complex and harder to implement; the end point of GRL links cannot be restricted within the Profile mechanism; there is no construct allowing for a GRL model evaluation.

7.1.2 Eclipse with EMF and GEF

jUCMNav [10] is a project focusing on a UCM modeling environment, which is based on a meta-model that is converted to EMF and GEF-based GUI. Since EMF provides a

framework for building EMF model editors, from a code level, jUCMNav has very powerful UCM modeling capabilities and a user friendly UI. The tool can also be extended to support GRL models in the future. The drawback of this approach of development is the longer time period it takes to learn and program this framework, compared with non-coding modeling editor development based on meta-modeling tools such as GME.

The following is a brief introduction to EMF and GEF.

Eclipse Modeling Framework (EMF)

EMF is a modeling framework and code generation facility for building tools and other applications based on a structured data model [4]. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor.

EMF consists of three parts:

- *EMF* – A meta-model and runtime support of models and other core framework;
- *EMF.Edit* – Framework including generic reusable classes for building editors for EMF models;
- *EMF.Codegen* – Generates everything needed to build a complete editor for an EMF model.

The EMF framework supports three levels of code generation:

- *Model* - provides Java interfaces and implementation classes for all the classes in the model, plus a factory and package implementation class.
- *Adapters* - generates implementation classes that adapt the model classes for editing and display.
- *Editor* - produces a properly structured editor that conforms to the recommended style for Eclipse EMF model editors and serves as a starting point from which to start customizing.

Graphical Editing Framework (GEF)

GEF is a framework that allows developers to take an existing application model and quickly create a rich graphical editor for it. It can easily be hooked to EMF meta-models.

The Pros and Cons

This approach, based on the powerful Eclipse Java API, provides user friendly UI and a framework of model-editor based on a meta-model. Java programming is easy to do, as compared with programming with COM API in GME. The ability to work with Eclipse plug-ins developed by a large group of users is also a benefit. The major disadvantage of this approach is the large amount of coding and the time it takes to learn to program with the framework. Although it needs COM programming in a GME based model analysis, a model editor in GME can be easily created via a graphical UI – no programming is needed.

7.1.3 Xactium's XMF-Mosaic

XMF-Mosaic is an integrated and extensible platform for building modeling tools and other types of applications such as programming languages [21]. It enables the creation of high fidelity models of business domain concepts using an integrated collection of model driven development standards including MOF, OCL, QVT and executable modeling. It also enables the rapid creation of languages and tools that target specific business domains and development processes.

It uses Snapshots to do a model instantiation – construct a model with diagram. No steps of interpreting the meta-model and generating paradigm (as in GME) before using it to create a model in GME.

It can also instantiate or test a model in a console window, using XOCL (Extensible Object Command Language) expressions. For example: "*X:=HelloWorld::HelloWorld()*" creates an instance of *HelloWorld* and assigns it to a variable X. "*X.text:="Hello""* assigns string "*Hello*" to X's attribute "text". This is a major advantage compared with our GRL editor.

It supports XML (textual) grammar definition, parsing and model population using XMF-Mosaic's grammar definition language, XBNF, which is a unique feature, as compared with other similar products that supports creation of model editors.

Other unique features include: support for user interface modeling capabilities using a family of user-interface description languages – Xtools, which creates a diagram editor (like GME paradigm) from a domain model (meta-model in GME); support for

model to model transformation - transform instances of one domain model into instances of another domain.

It supports a built-in Java generator that generates Java codes from models. However, one of the important aspects of creating a model editor – visual representation of the entities in a model, is not described in the article [16].

7.1.4 Organization Modeling Environment

OME is a general, goal-oriented and/or agent-oriented modeling and analysis tool [11]. It is developed by the Knowledge Management Lab at the University of Toronto. A domain specific modeling environment is called "framework" in OME. OME currently supports the following frameworks:

- *i** - The *i** framework proposes an agent-oriented approach to requirements engineering centering on the intentional characteristics of the agent.
- *NFR* – Non-Functional Requirements
- *GRL* – Goal-Oriented Requirements Language

The GRL framework provides a Java based API for the user to access the content of the model (also called *underlying knowledge base* that stores the semantics of the frameworks) and to extend the framework.

Compared with our GRL editor, support for actor concept in OME and a Java API that enables customized plug-in development, are the major benefits. It supports an easy to use GRL model evaluation feature. But it uses proprietary technologies that is not open source (TELOS database) [18]. Other issues or disadvantages with OME are: Object types and relationships in framework are specified in textual form, vs. visual editing of meta-model in GME; Structure of the model entities is "flat". No hierarchy of model entities such as a Model object and its contained Part objects in GME. No mechanism for entity reuse, such as Reference and Set in GME that enables one definition of entity being used and shared by multiple models.

7.2. Improvements to the GRL Meta-model

In the simplified GRL meta-model, Actor Boundary element is not included. An actor can own a group of elements in a GRL model. Such a concept can be implemented by a Set in GME. Set is the GME concept that is recommended for situations in which an object has to be associated with a relatively large number of neighbouring objects in a diagram. These objects are called the "members" of the set. GME provides a convenient way to assign objects to different Sets in a model. A model contains multiple Sets. Sets can also be used to implement any aggregation relationship in a model in the future.

OCL constraints are not used in the current GRL editor. GME supports a “customized” OCL – *MCL*, which is a MGA constraint language that is fully compliant with OCL 1.4 specification, with MGA-specific additions. A future complete GRL editor will use OCL to specify more complex constraints.

Other link types, such as Correlation link, Decomposition, Means-End and Dependency, are not implemented in the current GRL editor.

7.3. Support for GRL Model Analysis

GRL model evaluation is not covered in this project. However, a brief study is done on the possibility of adding GRL model evaluation function to the GRL Editor in the future. The study shows that GME provides a mechanism for developing customized model interpreters. The model interpreter is a COM server, or GME add-in, that is based on programming with the MGA COM library. Such an interpreter can be invoked with a menu item in GME. The interpreter can perform an object traversal, starting from the root node object, doing constraint checking, code generation, and satisfaction level calculation based on certain evaluation algorithm.

As described in section 2.2.7, two higher-level component interfaces have been provided by GME, which facilitates an interpreter development. For more information about this type of development, see [2].

Chapter 8. Conclusions

This project developed a GRL Editor based on a simplified version of a GRL meta-model. The GRL Editor can generate GRL models with proper visual presentation. The GRL Editor can be extended and the backward compatibility of an existing model is guaranteed under several conditions. Since a full scale GRL meta-model will most likely introduce more types of entities, further study on the extensibility of this GRL Editor is needed in the future. Model evaluation is another important topic that is not covered in this project. Study on the GME's concept of Interpreter shows that a COM server can be developed to perform an automatic evaluation of a GRL model. Though, COM programming in C++ is a demanding technique to be considered for a small scale project.

8.1. Contributions

- Study on the functionalities of GME and the feasibility of using GME to develop a GRL editor.
- Design of GRL editor via a GME-based meta-model
- GRL editor visualization enhancement via COM programming on the decorators
- Meta-model evolution experiments that study the effects of meta-model changes on the existing models.

8.2. Future Work

Future work related to the development of this GRL model editor based on a complete GRL meta-model will need to add other types of concepts that are mentioned in section 7.2, implement some complex constraints via OCL, implement a model interpreter to perform a model evaluation, and possibly some enhancement work to the visual representation of the GRL editor.

Import a GRL model into jUCMNav can be another major work item, which requires proper export of a GRL model from GME.

References

[1] Amyot, D.: Introduction to the User Requirements Notation: Learning by Example. In: *Computer Networks*, 42(3), 285-301, 21 June 2003.

[2] Bakay, A.: *The MGA library*. ISIS, Vanderbilt University http://www.cs.virginia.edu/~pnn7f/vest/docs/mgalib.pdf. Accessed May 2005.

[3] Chung, L., Nixon, B.A., Yu, E., & Mylopoulos, J.: *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishers, Dordrecht, USA, 2000.

[4] Eclipse: Graphical Modeling Framework (GMF), http://www.eclipse.org/gmf/

[5] IBM: *Rational Software Architecture*, http://www-306.ibm.com/software/awdtools/architect/swarchitect/

[6] Institute for Software Integrated Systems: *The Generic Modeling Environment (GME)*, 2004. http://www.isis.vanderbilt.edu/Projects/gme/

[7] ITU-T – International Telecommunications Union: Recommendation Z.150 (02/03), User Requirements Notation (URN) – Language Requirements and Framework. Geneva, Switzerland, 2003.

[8] ITU-T, URN Focus Group: *Draft Rec. Z.151 – GRL: Goal-oriented Requirement Language (GRL)*. Geneva, Switzerland, Sept. 2003. http://www.UseCaseMaps.org/urn/

[9] Janmohamed, J: *Expressing Goal-oriented Requirement Language in UML 2.0*. CSI 4900 project report, University of Ottawa, April 2005.

[10] jUCMNav: http://jkealey.shade.ca:82/twiki/bin/view/ProjetSEG/WebHome

[11] Knowledge Management Lab, University of Toronto: *Organization Modeling Environment*, http://www.cs.toronto.edu/km/ome/

[12] Kealey, J., Tremblay, E., Daigle, J.-P., McManus, J., Clift-Noël, O., and Amyot, D.: jUCMNav: une nouvelle plateforme ouverte pour l'édition et l'analyse de modèles UCM. To appear in: *Nouvelles TEchnnologies de la RÉpartition (NOTERE'05)*, Gatineau, Canada, August 2005.

[13] Lamsweerde, A.v.: Requirements Engineering in the Year 00: A Research Perspective. In: *Proc. of 22nd Intl Conference on Software Engineering (ICSE)*. Limerick, Ireland, ACM press, 2000.

[14] Liu, L., & Yu, E.: Designing Information Systems in Social Context: A Goal and Scenario Modelling Approach. *Information Systems (Journal)*, Vol.29, No.2. 2003. http://www.cs.toronto.edu/~liu/publications/

[15] OMG – Object Management Group (2003). *Unified Modeling Language Specification (UML)*, version 1.5, March 2003. http://www.omg.org/uml/

[16] Telelogic: *Telelogic TAU*, http://www.telelogic.com/products/tau/

[17] University of Toronto: *GRL Ontology*, http://www.cs.toronto.edu/km/GRL/

[18] University of Toronto: *Telos: Representing Knowledge about Information Systems*, http://www.cs.toronto.edu/~jm/2507S/Notes04/Telos.pdf

[19] URN Focus Group: *Draft Rec. Z.151 – Goal-oriented Requirement Language (GRL)*. Geneva, Switzerland, Sept. 2003. http://www.UseCaseMaps.org/urn/. See also http://www.cs.toronto.edu/km/GRL/

[20] Weiss, M., & Amyot, D.: Designing and Evolving Business Models with URN. *Montreal Conference on eTechnologies (MCeTech)*, Montréal, Canada, January 2005.

[21] Xactium Limited: XMF-Mosaic Getting Started Guide, Version 1.0, July 2005

[22] Yu, E.: Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering. *3rd IEEE Int. Symp. on Requirements Engineering (RE'97)*, Washington, USA, 226-235, 1997.

[23] Yu, E., & Liu, L.: *Organization Modelling Environment* (OME), 2000. http://www.cs.toronto.edu/km/ome/

[24] Yu, E., & Mylopoulos, J.: Why goal-oriented requirements engineering. *Proceedings of the 4th REFSQ*, Pisa, Italy, 15–22, 1998.